

PLANARITY TESTING BY PATH ADDITION

A THESIS SUBMITTED TO
THE UNIVERSITY OF KENT AT CANTERBURY
IN THE SUBJECT OF COMPUTER SCIENCE
FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY.

By
Martyn G. Taylor
12th September 2008
Revised 9th May 2011

Contents

List of Tables	xi
List of Figures	xx
Abstract	xxi
1 Introduction	1
1.1 Motivation	1
1.2 Thesis Aims	3
1.3 Thesis Outline	3
2 A History of Planarity Testing and Embedding	4
2.1 Euler Characteristic – 1752	6
2.2 Trémaux’s (Depth-First Search) Algorithm – 1882	7
2.2.1 Tarry’s Algorithm – 1895	7
2.2.2 Trémaux Trees	8
2.3 Jordan Curve Theorem – 1887	9
2.3.1 Historical Note on the Correctness of Jordan’s Proof	10
2.4 Kuratowski’s Theorem – 1930	11
2.5 Hassler Whitney – 1932 & 1933	12
2.6 Lempel, Even, and Cederbaum’s $O(\mathcal{V}^2)$ Vertex Addition Planarity Test – 1967 .	13
2.6.1 st -ordering	13
2.6.2 Bush Form	13
2.7 Wagner’s Conjecture (Theorem) - 1970	15

2.8	Hopcroft and Tarjan’s Path Addition Planarity Test – 1974	16
2.8.1	Example of H&T’s Algorithm	17
2.8.2	Extensions to H&T’s algorithm	19
2.8.3	“Complexity” of H&T’s algorithm	19
2.9	Booth and Lueker’s Vertex Addition Planarity Test – 1976	21
2.10	Shih-Hsu – Edge Addition Planarity Test 1999	22
2.11	Boyer-Myrvold Planarity Test	23
2.12	De Fraysseix and Rosenstiehl’s Trémaux Tree Planarity Characterisation and Test – 1982-2008	24
2.13	Planarity Algorithms via PQ-Trees – 2008	26
2.14	Conclusions	26
3	Trémaux Trees	27
3.1	Trémaux Tree Definitions	27
4	Planarity Testing	31
4.1	Segments of a Trémaux Tree	31
4.1.1	Ordering the Segments	38
4.1.2	Varying the Order of Incomparable TT-Precedence Edges	39
4.2	Planar Embedding	40
4.2.1	Embeddings, Faces and Regions	40
4.2.2	Segment Classes	41
4.2.3	Defining Segments As “Having An Embedding”	42
4.2.4	Defining Sides to the Root Segment	43
4.2.5	Defining the Inside and Outside Relative to Non-Root Segments	44
4.2.6	Embedding Class 1, 2 or 3 Segments	45
4.2.7	Embedding Class 4 Segments	46
4.3	Permutations of Embeddings	60
4.3.1	Handling Flippable Segments	61
4.3.2	Handling Permutable Segments	61
4.3.3	Handling the Special Case (Permutable and Flippable and ...)	62

4.3.4	Handling Reversible Segments	64
4.3.5	Handling Rotatable Segments	64
4.4	Generating a Cyclic Edge Order	68
4.4.1	Initial Cyclic Edge Order (Solely Within a Segment)	68
4.4.2	Generating a Cyclic Edge Order for a Biconnected Component	69
5	The Algorithm	72
5.1	Depth First Search	73
5.2	Expected Data Structures	74
5.3	Trémaux Tree Generation Algorithm	75
5.4	Segment Generation Algorithm	79
5.5	Planarity Testing and Permutation Generation Algorithm	82
5.6	Reordering Permutations	90
5.6.1	Steinhaus-Johnson-Trotter Permutation Algorithm	90
5.7	Edge Order Generation Algorithm	98
5.8	Planarity Testing Conclusions	100
6	Benchmarking Methodology	101
6.1	What is software performance benchmarking?	101
6.2	Factors affecting the Robustness of a Software Benchmark	102
6.3	Mitigating for Software Factors during Benchmarking	104
6.3.1	Clock Resolution	104
6.3.2	Code Warm-Up	106
6.3.3	Garbage Collection	112
6.3.4	Logging Results	118
6.4	Analysis Techniques to Reduce the Impact of Unmitigated Factors	119
6.4.1	Robust Curve Fitting Techniques	119
6.5	Benchmarking Methodology Conclusions	122
7	Empirical Testing	124
7.1	Implementing the Planarity Testing Algorithm	125

7.1.1	Aims and Limitations of the Planarity Testing Algorithm	125
7.2	Empirical Testing Plan	126
7.3	Test Graphs	129
7.3.1	Triangular Prism Graph	131
7.3.2	Planar Wheel Graph	132
7.3.3	Ordered Face Trisection Graph	133
7.3.4	Random Face Trisection Graphs	134
7.3.5	Permutable, Flippable Graphs	134
7.3.6	Sub-Graph Generation via Non-Bridge Edge Deletion	135
7.3.7	Comparison of Test Graph's Properties	137
7.3.8	Scalability of Test Graphs	138
7.4	Empirical Testing using Different Root Vertices	143
7.4.1	Conclusions for Varying Root Vertex for a Constant Graph Size and Structure	150
7.5	Empirical Testing on Maximal-Planar Graphs of Varying Size	151
7.5.1	Conclusions for Empirical Testing on Maximal-Planar Graphs of Varying Size	158
7.6	Empirical Testing on Graphs of Constant Vertex-Size and Varying Edge-Size . .	159
7.7	Empirical Testing on Permutable, Flippable Graphs of Varying Size	164
7.8	Conclusions	165
8	Conclusions	167
8.1	Future Work	169
A	Empirical Testing Figures	171
A.1	Variations in Root Vertex for Constant Graph Size and Type	172
A.1.1	Triangular Prism Graphs	172
A.1.2	Planar Wheel Graphs	175
A.1.3	Ordered Face Trisection Graphs	178
A.1.4	Random Face Trisection Graphs	182
A.2	Variation in Graph Size for Constant Graph Type and Root Vertex	185
A.2.1	Triangular Prism Graphs (42-4500 Vertices)	185

A.2.2	Triangular Prism Graphs (500-15000 Vertices)	187
A.2.3	Planar Wheel Graphs (42-4500 Vertices)	190
A.2.4	Planar Wheel Graphs (500-15000 Vertices)	195
A.2.5	Ordered Face Trisection Graphs (42-4500 Vertices)	198
A.2.6	Ordered Face Trisection Graphs (500-15000 Vertices)	200
A.2.7	Random Face Trisection Graphs (42-4500 Vertices)	202
A.2.8	Random Face Trisection Graphs (500-15000 Vertices)	205
A.3	Variations in Number of Edges for Sub-Graphs of Constant Graph Size and Type	208
A.3.1	Triangular Prism Graphs (14999 Vertices)	208
A.3.2	Planar Wheel Graphs (15000 Vertices)	212
A.3.3	Ordered Face Trisection Graphs (14999 Vertices)	215
A.3.4	Random Face Trisection Graphs (14999 Vertices)	219
A.4	Variations in Number of Edges for Sub-Graphs of Constant Graph Size and Type – Effects on Numbers of Conflicts and Mean Segment Depth	222
A.4.1	Triangular Prism Graphs	222
A.4.2	Planar Wheel Graphs	224
A.4.3	Ordered Face Trisection Graphs	226
A.4.4	Random Face Trisection Graphs	229
A.5	Variations in Size of Permutable, Flippable Graphs	231
B	Configuration used for Empirical Testing	234
C	Clock Resolution Testing	235
C.1	Clock Resolution Test Code	235
C.2	Clock Resolution Raw Data	236
D	Java Source Code	240
D.1	Data Structure Files	240
D.1.1	mgtaylor.datastructures.AccessibleUnmodifiableLinkedList	240
D.1.2	mgtaylor.datastructures.LinkedList	242
D.1.3	mgtaylor.datastructures.PermutableUnmodifiableLinkedList	248
D.1.4	mgtaylor.datastructures.UnmodifiableLinkedList	261

D.2	Graph Files	268
D.2.1	mgtaylor.graph.Block	268
D.2.2	mgtaylor.graph.Edge	273
D.2.3	mgtaylor.graph.Graph	279
D.2.4	mgtaylor.graph.GraphEvent	296
D.2.5	mgtaylor.graph.GraphListener	297
D.2.6	mgtaylor.graph.GraphObject	297
D.2.7	mgtaylor.graph.Path	297
D.2.8	mgtaylor.graph.PathSegment	301
D.2.9	mgtaylor.graph.Segment	303
D.2.10	mgtaylor.graph.SegmentPermutation	315
D.2.11	mgtaylor.graph.Vertex	317
D.3	Graph Type Files	327
D.3.1	mgtaylor.graph.graphTypes.OrderedFaceTrisectionGraph	327
D.3.2	mgtaylor.graph.graphTypes.PlanarMultiWheelGraph	329
D.3.3	mgtaylor.graph.graphTypes.PlanarWheelGraph	331
D.3.4	mgtaylor.graph.graphTypes.PermutableFlippableGraph	332
D.3.5	mgtaylor.graph.graphTypes.RandomMaximalPlanarGraphGenerator	333
D.3.6	mgtaylor.graph.graphTypes.ReduceableGraphGenerator	334
D.3.7	mgtaylor.graph.graphTypes.ReduceableBiconnectedGraphGenerator	338
D.3.8	mgtaylor.graph.graphTypes.TriangularPrismMaximalPlanarGraph	342
D.4	Empirical Testing Files	344
D.4.1	mgtaylor.graph.test.PlanaritySpeedTestViewer	344
D.4.2	mgtaylor.graph.test.parsers.GraphExtractorLogParser	352
D.4.3	mgtaylor.graph.test.parsers.GraphStatsLogParser	353
D.4.4	mgtaylor.graph.test.parsers.NewGCLogParser	355
D.4.5	mgtaylor.graph.test.utilities.ConcatenateFiles	359
D.4.6	mgtaylor.graph.test.utilities.SplitLogFiles	360
D.5	GUI Files	363
D.5.1	mgtaylor.display.AngleHighlightView	363

D.5.2	mgtaylor.display.Arrow	366
D.5.3	mgtaylor.display.DraggableView	367
D.5.4	mgtaylor.display.JIntegerTextField	372
D.5.5	mgtaylor.display.PlanarityHandler	373
D.5.6	mgtaylor.display.PlanarityViewer	374
D.5.7	mgtaylor.display.View	381
D.5.8	mgtaylor.display.data.BlockDataView	391
D.5.9	mgtaylor.display.data.ButtonColumn	393
D.5.10	mgtaylor.display.data.DataView	395
D.5.11	mgtaylor.display.data.PermutationDataView	395
D.5.12	mgtaylor.display.data.SegmentDataView	398
D.5.13	mgtaylor.display.data.VertexDataView	400
D.5.14	mgtaylor.display.data.event.JTableButtonEvent	402
D.5.15	mgtaylor.display.data.event.JTableButtonHandler	402
D.5.16	mgtaylor.display.data.event.JTableButtonListener	403
D.5.17	mgtaylor.display.event.GraphDisplayEvent	403
D.5.18	mgtaylor.display.event.GraphDisplayHandler	404
D.5.19	mgtaylor.display.event.GraphDisplayListener	404
D.5.20	mgtaylor.layout.ConstantMovementMap	404
D.5.21	mgtaylor.layout.MovementMap	405
D.5.22	mgtaylor.layout.OrderedVertexLayout	405
D.5.23	mgtaylor.layout.PermutableFlippableLayout	406
D.5.24	mgtaylor.layout.PlanarWheelLayout	406
D.5.25	mgtaylor.layout.RandomVertexLayout	408
D.5.26	mgtaylor.layout.TriangularPrismMaximalPlanarLayout	408
D.5.27	mgtaylor.layout.VertexLayout	409
D.5.28	mgtaylor.layout.event.GraphLayoutEvent	411
D.5.29	mgtaylor.layout.event.GraphLayoutHandler	412
D.5.30	mgtaylor.layout.event.GraphLayoutListener	412
D.5.31	mgtaylor.planarity.event.PlanarOrderEvent	412

D.5.32	mgtaylor.planarity.event.PlanarOrderHandler	413
D.5.33	mgtaylor.planarity.event.PlanarOrderListener	413
D.5.34	mgtaylor.tools.Trigonometry	413
D.6	L ^A T _E X Segment Hierarchy Generation	419
D.6.1	mgtaylor.latex.SegmentHierarchies	419
E	Matlab	422
E.1	Matlab Code to Load Data	422
E.2	Matlab Code To Generate Figures	423
E.3	Matlab Code To Generate Tables	431
E.4	Matlab Functions	432
E.4.1	appendPercentiles.m	432
E.4.2	filldata.m	432
E.4.3	fitdata2.m	433
E.4.4	fitdatatoarray.m	437
E.4.5	loadAndSplitGraphData.m	440
E.4.6	loadGraphData.m	441
E.4.7	performKS2Tests.m	443
E.4.8	plotGraphData.m	445
E.4.9	PlotNumberOfElements.m	453
E.4.10	PlotShowCodeWarmUp.m	455
E.4.11	processOptions.m	457
E.4.12	removeoutliers.m	458
	General Mathematical Notations	460
	Graph Notations	462
	Glossary	464
	Graphical Notations	471
	Bibliography	472

List of Tables

5.6.1	23 rd Steinhaus-Johnson-Trotter permutation of the sequence $\langle A, B, C, D, E \rangle$. . .	96
6.3.1	Summary of Clock Resolution Times	105
6.3.2	Dissimilarity between Datasets Affected and Unaffected by Garbage Collection	116
6.4.1	Comparison of Ordinary and Iteratively Re-Weighted Least Squares	122
7.3.1	Vertex Degree Frequencies in a Triangular Prism Graph of size N	131
7.3.2	Vertex Degree Frequencies in a Planar Wheel Graph of size N	132
7.3.3	Comparison of Test Graph Properties	137
7.4.1	Mean Absolute Error for Best-Fit Curves to Triangular Prism Results with Varying Root Vertices	145
7.4.2	Mean Absolute Error for Best-Fit Curves to Planar Wheel Results with Varying Root Vertices	146
7.4.3	Mean Absolute Error for Best-Fit Curves to Ordered Face Trisection Results with Varying Root Vertices	147
7.4.4	Mean Absolute Error for Best-Fit Curves to Random Face Trisection Results with Varying Root Vertices	150
7.5.1	Mean Absolute Error for Best-Fit Curves to Triangular Prism (42-4500 Vertices) Results	152
7.5.2	Mean Absolute Error for Best-Fit Curves to Triangular Prism (500-15000 Vertices) Results	152
7.5.3	Mean Absolute Error for Best-Fit Curves to Planar Wheel (42-4500 Vertices) Results	153
7.5.4	Mean Absolute Error for Best-Fit Curves to Planar Wheel (Up to 4500 Vertices) Results After Removal of Tiny Graph Sizes	154
7.5.5	Mean Absolute Error for Best-Fit Curves to Planar Wheel (500-15000 Vertices) Results	154
7.5.6	Mean Absolute Error for Best-Fit Curves to Ordered Face Trisection (42-4500 Vertices) Results	155

7.5.7	Mean Absolute Error for Best-Fit Curves to Ordered Face Trisection (500-15000 Vertices) Results	156
7.5.8	Mean Absolute Error for Best-Fit Curves to Random Face Trisection (up to 4500 Vertices) Results	157
7.5.9	Mean Absolute Error for Best-Fit Curves to Random Face Trisection (up to 15000 Vertices) Results	157
7.6.1	Mean Absolute Error for Best-Fit Curves to Biconnected Triangular Prism (TP), Planar Wheel (PW), Ordered Face Trisection (OFT) and Random Face Trisection (RFT) Sub-Graph Results (14999 Vertices, ~30000-45000 Edges, Various Root Vertices)	159
7.6.2	Mean Absolute Error for Best-Fit Curves to Ordered Face Trisection Sub-Graph Results (14999 Vertices, Varying Edges)	160
7.6.3	Mean Absolute Error for Best-Fit Curves to Triangular Prism Sub-Graph Results (14999 Vertices, Varying Edges)	161
7.7.1	Mean Absolute Error for Best-Fit Curves to Permutable Flippable Graph Results (1002-15002 Vertices)	164

List of Figures

2.3.1	Rays Intersecting a Jordan Curve	9
2.3.2	Three Non-Intersecting Jordan Arcs With Shared Endpoints	9
2.4.1	K_5 and $K_{3,3}$ Non-Planar Graphs	11
2.5.1	A Graph (blue/black) and its Dual (red)	12
2.5.2	Example of a <i>Whitney Flip</i>	12
2.6.1	Sample Planar Graph and corresponding Bush Forms $\mathcal{B}_2 \dots \mathcal{B}_5$	14
2.6.2	Non-Planar Graphs K_5 and $K_{(3,3)}$ and their corresponding Bush Forms. . . .	14
2.7.1	Example of a $K_{3,3}$ Graph Minor	15
2.8.1	Paths (Green and Blue) on <i>Left</i> and <i>Right</i> of Cycle (Red) Formed by Path ($x \rightarrow y \rightarrow z$).	17
2.8.2	Example Graph and Sample Path Taken by DFS	17
2.8.3	Disjoint Paths Generated by Second Phase of H&T's Planarity Test	17
2.8.4	Example Embedding of Figure 2.8.2	18
2.8.5	Contents of Stacks during Embedding of Figure 2.8.2	18
2.8.6	Second Example of Disjoint Paths (Numbered in Embedding Order) Generated by Second Phase of H&T's Planarity Test	20
2.8.7	Second Example – Embedding of Successive Paths (Contents of Coloured Regions Correspond to Contents of Stacks in Figure 2.8.8)	20
2.8.8	Second Example – Contents of Left-Right Stacks Corresponding to Successive Embeddings Showing Lack of Correlation Between Path's Stack and Embed- ding Region	20
2.10.1	Example of the Generation of C-nodes and Their Concatenation	22
2.12.1	De Fraysseix and Rosenstiehl's Planarity Characterisation on Trémaux Trees .	24
3.1.1	Example Graph with Edges Labelled by Depth-First Search Pre-Order (Tree edges are thick, cotree edges are dashed).	28

3.1.2	Trémaux Tree of Example Graph (Figure 3.1.1) where each connection in the tree represents a $\prec$ relationship	28
3.1.3	Reachable vertices and edges (red) from edge $(L \xrightarrow{T} M)$ for the Trémaux Tree (Figure 3.1.2) of Example Graph	29
3.1.4	(a) Edges outbound from the same vertex in TT-precedence order from the three outermost edges with (incomparable) lowest precedence to the innermost with highest precedence. (b) Two sets (red and green) of incomparable precedence edges outbound from the same vertex – the green edges have higher precedence than the red edges.	30
4.1.1	Nine segments (shown as coloured groups) which comprise the Trémaux Tree of Example Graph (Figure 3.1.1)	34
4.1.2	Examples of Segment Classes Based on Tail Vertex Precedence (Head Vertex in Red)	35
4.1.3	Segments from Figure 4.1.1 Numbered in $<^s$ Order	38
4.1.4	Segment Parent-Child Relationships from Figure 4.1.1	38
4.2.1	Stereographic Projection Mapping Between the Surface of a Unit Sphere $(P = (1, \lambda, \varphi))$ and the $\mathbb{R}^2$ Plane $(Q = (\cot(\lambda/2), \varphi))$	41
4.2.2	Examples of Segment Classes Based on Tail Vertex Precedence (Head Vertex in Red)	41
4.2.3	Inside (Yellow) and Outside (Brown) Adjacent Regions of Different Classes of Root Segment (Black/Blue) as Defined by Location of Branching Path/Cycle	43
4.2.4	Examples of Inside (Yellow) and Outside (Brown) Regions Formed by Embedding Different Segment Classes	44
4.2.5	Example of Possible Embeddings of (Red) Segment Rotating About Head Vertex Into Different Adjacent Faces	45
4.2.6	Example Clockwise Embedding of Class 2 Segment s and Reverse Anti-Clockwise Embedding	45
4.2.7	Embeddings of Segments s_i, s_j & s_k Where $s_i <^s s_j <^s s_k$ and s_i is the Parent of s_k and is an Ancestor of s_j (Sub-Cases of Case 2, Pg. 46) and Showing Inside (Yellow) and Outside(Brown) Faces Defined by s_j	48
4.2.8	Permutable Segments s_k and $s_{j'}$	51
4.2.9	Segments of Sample Graph Numbered in $<^s$ Order	58
4.2.10	Embedding of Segments of Sample Graph	58
4.2.11	Contents of Blocks during Embedding From Figure 4.2.10	58
4.3.1	Possible Embeddings of a Class 4 Child Segment Attached to a Non-Root Class 3 Segment Where Both Segment Have A Common Tail Vertex	60
4.3.2	Degenerate Embedding Case	61

4.3.3	Example Graph with 3 <i>Flippable</i> Blocks and 2 <i>Permutable, Flippable, Non-Conflicting</i> Segments Including the <i>Static</i> Child Segment	63
4.3.4	All Permutations of Example Graph in Figure 4.3.3	63
4.3.5	Four Possible Embeddings (a-d) Within the Outside Face and When <i>Flipped</i> (e) a Single Embedding Within the Inside Face	65
4.3.6	Example Permutations of Class 3 Segments	66
4.4.1	Three Non-Intersecting Jordan Arcs With Shared Endpoints	68
4.4.2	Segments With Clockwise (Red) and Anti-Clockwise (Blue) Cyclic Direction Connecting to Parent With Clockwise (Black) or Anti-Clockwise (Black and Grey) Cyclic Direction	69
4.4.3	Angles From Parent & Stem Edges To Segments With Increasing $<^s$ Precedence	69
4.4.4	Angles From Parent & Neighbouring Edges To Segments With Increasing $<^s$ Precedence	71
5.6.1	Graphical Representation of Steinhaus-Johnson-Trotter Permutation Algorithm for One (a) to Four (d) Elements Showing how the Pattern of Permutations Generated Builds on the Previous.	91
6.3.1	Code Warm-Up for 1250 Vertex Triangular Graph (Root = 1)	109
6.3.2	Code Warm-Up for 1250 Vertex Triangular Graph (Root = 2)	110
6.3.3	Example comparing <i>ECDFs</i> of samples drawn from two normal distributions.	115
6.3.4	Number of Data Points (Out of 100) Unaffected By GC for Different Graph Types (Root = 1st Vertex)	117
6.4.1	Least Squares Curve Fitting	120
6.4.2	Iteratively Re-Weighted Least Squares Curve Fitting	121
7.3.1	Examples of Triangular Prism Graphs	131
7.3.2	Examples of Planar Wheel Graphs	132
7.3.3	Examples of Ordered Face Trisection Graphs	133
7.3.4	Examples of Permutable, Flippable Graphs	134
7.3.5	Segment Hierarchy for Triangular Prism Graphs (Root = 1 st Vertex)	140
7.3.6	Segment Hierarchy for Triangular Prism Graphs (Root = N th Vertex)	140
7.3.7	Segment Hierarchy for Planar Wheel Graphs (Root = 1 st Vertex)	141
7.3.8	Segment Hierarchy for Planar Wheel Graphs (Root = N th Vertex)	141
7.3.9	Segment Hierarchy for Ordered Face Trisection Graphs (Root = 1 st Vertex) .	142
7.3.10	Segment Hierarchy for Ordered Face Trisection Graphs (Root = N th Vertex) .	142

7.4.1	Execution Times for Varying Root Vertices of 1000 Vertex Triangular Prism Graph	143
7.4.2	Execution Times for Varying Root Vertices of 1000 Vertex Triangular Prism Graph (98 th %ile Data, Mapped Root ID)	144
7.4.3	Execution Times for Varying Root Vertices of 1000 Vertex Planar Wheel Graph	145
7.4.4	Execution Times for Varying Root Vertices of 1000 Vertex Ordered Face Trisection Graph	147
7.4.5	Execution Times for Varying Root Vertices of 1000 Vertex Random Face Trisection Graph	148
7.4.6	Two Overlaid Result-Sets for Execution Times for Varying Root Vertices of the Same 1000 Vertex Random Face Trisection Graph	149
7.4.7	Execution Times for Varying Root Vertices of 1000 Vertex Random Face Trisection Graph (Results for Execution Times < 8.1ms)	149
7.5.1	Triangular Prism Graph (42-4500 Vertices, 1 st Vertex Root)	151
7.5.2	Planar Wheel Graph (42-4500 Vertices, 1 st Vertex Root)	153
7.5.3	Ordered Face Trisection Graph (42-4500 Vertices, 1 st Vertex Root)	155
7.5.4	Random Face Trisection Graph (42-4500 Vertices, 1 st Vertex Root)	156
7.6.1	Ordered Face Trisection Graph (14999 Vertices, ~30000-43000 Edges, $\frac{1}{4}N^{\text{th}}$ Vertex Root)	160
7.6.2	Biconnected Sub-Graphs of Triangular Prism Graph (14999 Vertices, ~30000-45000 Edges (Split at 40000 Edges), $\frac{1}{4}N^{\text{th}}$ Vertex Root)	161
7.6.3	Biconnected Sub-Graphs of Triangular Prism Graph (14999 Vertices, ~30000-45000 Edges (Split at 40000 Edges), $\frac{1}{4}N^{\text{th}}$ Vertex Root)	161
7.6.4	Mean Segment Depth for Biconnected Sub-Graphs of Triangular Prism Graph (14999 Vertices, ~30000-45000 Edges, $\frac{1}{4}N^{\text{th}}$ Vertex Root)	162
A.1.1	Execution Times for Varying Root Vertices of 250 Vertex Triangular Prism Graph	172
A.1.2	Execution Times for Varying Root Vertices of 500 Vertex Triangular Prism Graph	173
A.1.3	Execution Times for Varying Root Vertices of 750 Vertex Triangular Prism Graph	173
A.1.4	Execution Times for Varying Root Vertices of 1250 Vertex Triangular Prism Graph	174
A.1.5	Execution Times for Varying Root Vertices of 1500 Vertex Triangular Prism Graph	174
A.1.6	Execution Times for Varying Root Vertices of 250 Vertex Planar Wheel Graph	175

A.1.7	Execution Times for Varying Root Vertices of 500 Vertex Planar Wheel Graph	175
A.1.8	Execution Times for Varying Root Vertices of 750 Vertex Planar Wheel Graph	176
A.1.9	Execution Times for Varying Root Vertices of 1250 Vertex Planar Wheel Graph	176
A.1.10	Execution Times for Varying Root Vertices of 1500 Vertex Planar Wheel Graph	177
A.1.11	Execution Times for Varying Root Vertices of 250 Vertex Ordered Face Trisection Graph	178
A.1.12	Execution Times for Varying Root Vertices of 500 Vertex Ordered Face Trisection Graph	179
A.1.13	Execution Times for Varying Root Vertices of 750 Vertex Ordered Face Trisection Graph	179
A.1.14	Execution Times for Varying Root Vertices of 1250 Vertex Ordered Face Trisection Graph	180
A.1.15	Execution Times for Varying Root Vertices of 1500 Vertex Ordered Face Trisection Graph	180
A.1.16	Execution Times for Varying Root Vertices of 2000 Vertex Ordered Face Trisection Graph	181
A.1.17	Execution Times for Varying Root Vertices of 250 Vertex Random Face Trisection Graph	182
A.1.18	Execution Times for Varying Root Vertices of 500 Vertex Random Face Trisection Graph	183
A.1.19	Execution Times for Varying Root Vertices of 750 Vertex Random Face Trisection Graph	183
A.1.20	Execution Times for Varying Root Vertices of 1250 Vertex Random Face Trisection Graph	184
A.1.21	Execution Times for Varying Root Vertices of 1500 Vertex Random Face Trisection Graph	184
A.2.1	Triangular Prism Graph (42-4500 Vertices, $\frac{1}{4}N^{\text{th}}$ Vertex Root)	185
A.2.2	Triangular Prism Graph (42-4500 Vertices, $\frac{1}{2}N^{\text{th}}$ Vertex Root)	186
A.2.3	Triangular Prism Graph (42-4500 Vertices, $\frac{3}{4}N^{\text{th}}$ Vertex Root)	186
A.2.4	Triangular Prism Graph (42-4500 Vertices, N^{th} Vertex Root)	187
A.2.5	Triangular Prism Graph (500-15000 Vertices, 1^{st} Vertex Root)	187
A.2.6	Triangular Prism Graph (500-15000 Vertices, $\frac{1}{4}N^{\text{th}}$ Vertex Root)	188
A.2.7	Triangular Prism Graph (500-15000 Vertices, $\frac{1}{2}N^{\text{th}}$ Vertex Root)	188
A.2.8	Triangular Prism Graph (500-15000 Vertices, $\frac{3}{4}N^{\text{th}}$ Vertex Root)	189
A.2.9	Triangular Prism Graph (500-15000 Vertices, N^{th} Vertex Root)	189

A.2.10	Planar Wheel Graph (42-4500 Vertices, $\frac{1}{4}N^{\text{th}}$ Vertex Root)	190
A.2.11	Planar Wheel Graph (42-4500 Vertices, $\frac{1}{2}N^{\text{th}}$ Vertex Root)	190
A.2.12	Planar Wheel Graph (42-4500 Vertices, $\frac{3}{4}N^{\text{th}}$ Vertex Root)	191
A.2.13	Planar Wheel Graph (42-4500 Vertices, $\frac{3}{4}N^{\text{th}}$ Vertex Root, 2 nd Run)	191
A.2.14	Planar Wheel Graph (42-4500 Vertices, N^{th} Vertex Root)	192
A.2.15	Planar Wheel Graph (750-4500 Vertices, 1 st Vertex Root)	192
A.2.16	Planar Wheel Graph (750-4500 Vertices, $\frac{1}{4}N^{\text{th}}$ Vertex Root)	193
A.2.17	Planar Wheel Graph (500-4500 Vertices, $\frac{1}{2}N^{\text{th}}$ Vertex Root)	193
A.2.18	Planar Wheel Graph (750-4500 Vertices, $\frac{3}{4}N^{\text{th}}$ Vertex Root)	194
A.2.19	Planar Wheel Graph (750-4500 Vertices, $\frac{3}{4}N^{\text{th}}$ Vertex Root, 2 nd Run)	194
A.2.20	Planar Wheel Graph (750-4500 Vertices, N^{th} Vertex Root)	195
A.2.21	Planar Wheel Graph (500-15000 Vertices, 1 st Vertex Root)	195
A.2.22	Planar Wheel Graph (500-15000 Vertices, $\frac{1}{4}N^{\text{th}}$ Vertex Root)	196
A.2.23	Planar Wheel Graph (500-15000 Vertices, $\frac{1}{2}N^{\text{th}}$ Vertex Root)	196
A.2.24	Planar Wheel Graph (500-15000 Vertices, $\frac{3}{4}N^{\text{th}}$ Vertex Root)	197
A.2.25	Planar Wheel Graph (500-15000 Vertices, N^{th} Vertex Root)	197
A.2.26	Ordered Face Trisection Graph (42-4500 Vertices, $\frac{1}{4}N^{\text{th}}$ Vertex Root)	198
A.2.27	Ordered Face Trisection Graph (42-4500 Vertices, $\frac{1}{2}N^{\text{th}}$ Vertex Root)	198
A.2.28	Ordered Face Trisection Graph (42-4500 Vertices, $\frac{3}{4}N^{\text{th}}$ Vertex Root)	199
A.2.29	Ordered Face Trisection Graph (42-4500 Vertices, N^{th} Vertex Root)	199
A.2.30	Ordered Face Trisection Graph (500-15000 Vertices, 1 st Vertex Root)	200
A.2.31	Ordered Face Trisection Graph (500-15000 Vertices, $\frac{1}{4}N^{\text{th}}$ Vertex Root)	200
A.2.32	Ordered Face Trisection Graph (500-15000 Vertices, $\frac{1}{2}N^{\text{th}}$ Vertex Root)	201
A.2.33	Ordered Face Trisection Graph (500-15000 Vertices, $\frac{3}{4}N^{\text{th}}$ Vertex Root)	201
A.2.34	Ordered Face Trisection Graph (500-15000 Vertices, N^{th} Vertex Root)	202
A.2.35	Random Face Trisection Graph (42-4500 Vertices, $\frac{1}{4}N^{\text{th}}$ Vertex Root)	202
A.2.36	Random Face Trisection Graph (42-4500 Vertices, $\frac{1}{2}N^{\text{th}}$ Vertex Root)	203
A.2.37	Random Face Trisection Graph (42-4500 Vertices, $\frac{3}{4}N^{\text{th}}$ Vertex Root)	203
A.2.38	Random Face Trisection Graph (42-4500 Vertices, N^{th} Vertex Root)	204
A.2.39	Random Face Trisection Graph (400-4500 Vertices, $\frac{3}{4}N^{\text{th}}$ Vertex Root)	204

A.2.40	Random Face Trisection Graph (500-15000 Vertices, 1 st Vertex Root)	205
A.2.41	Random Face Trisection Graph (500-15000 Vertices, $\frac{1}{4}N^{\text{th}}$ Vertex Root)	205
A.2.42	Random Face Trisection Graph (500-15000 Vertices, $\frac{1}{2}N^{\text{th}}$ Vertex Root)	206
A.2.43	Random Face Trisection Graph (500-15000 Vertices, $\frac{3}{4}N^{\text{th}}$ Vertex Root)	206
A.2.44	Random Face Trisection Graph (500-15000 Vertices, N^{th} Vertex Root)	207
A.2.45	Random Face Trisection Graph (750-15000 Vertices, $\frac{3}{4}N^{\text{th}}$ Vertex Root)	207
A.3.1	Triangular Prism Graph (14999 Vertices, ~30000-45000 Edges, 1 st Vertex Root)	208
A.3.2	Triangular Prism Graph (14999 Vertices, ~30000-45000 Edges, $\frac{1}{4}N^{\text{th}}$ Vertex Root)	209
A.3.3	Triangular Prism Graph (14999 Vertices, ~30000-45000 Edges, $\frac{1}{4}N^{\text{th}}$ Vertex Root, 2 nd Run)	209
A.3.4	Triangular Prism Graph (14999 Vertices, ~30000-45000 Edges, $\frac{1}{2}N^{\text{th}}$ Vertex Root)	210
A.3.5	Triangular Prism Graph (14999 Vertices, ~30000-45000 Edges, $\frac{3}{4}N^{\text{th}}$ Vertex Root)	210
A.3.6	Triangular Prism Graph (14999 Vertices, ~30000-45000 Edges, N^{th} Vertex Root)	211
A.3.7	Planar Wheel Graph (15000 Vertices, ~30000-45000 Edges, 1 st Vertex Root)	212
A.3.8	Planar Wheel Graph (15000 Vertices, ~30000-45000 Edges, $\frac{1}{4}N^{\text{th}}$ Vertex Root)	212
A.3.9	Planar Wheel Graph (15000 Vertices, ~30000-45000 Edges, $\frac{1}{2}N^{\text{th}}$ Vertex Root)	213
A.3.10	Planar Wheel Graph (15000 Vertices, ~30000-45000 Edges, $\frac{3}{4}N^{\text{th}}$ Vertex Root)	213
A.3.11	Planar Wheel Graph (15000 Vertices, ~30000-45000 Edges, N^{th} Vertex Root)	214
A.3.12	Ordered Face Trisection Graph (14999 Vertices, ~30000-45000 Edges, 1 st Vertex Root)	215
A.3.13	Ordered Face Trisection Graph (14999 Vertices, ~30000-45000 Edges, $\frac{1}{4}N^{\text{th}}$ Vertex Root)	216
A.3.14	Ordered Face Trisection Graph (14999 Vertices, ~30000-45000 Edges, $\frac{1}{4}N^{\text{th}}$ Vertex Root, 2 nd Run)	216
A.3.15	Ordered Face Trisection Graph (14999 Vertices, ~30000-45000 Edges, $\frac{1}{2}N^{\text{th}}$ Vertex Root)	217
A.3.16	Ordered Face Trisection Graph (14999 Vertices, ~30000-45000 Edges, $\frac{3}{4}N^{\text{th}}$ Vertex Root)	217
A.3.17	Ordered Face Trisection Graph (14999 Vertices, ~30000-45000 Edges, N^{th} Vertex Root)	218
A.3.18	Random Face Trisection Graph (14999 Vertices, ~30000-45000 Edges, 1 st Vertex Root)	219

A.3.19	Random Face Trisection Graph (14999 Vertices, ~ 30000 -45000 Edges, $1/4N^{\text{th}}$ Vertex Root)	220
A.3.20	Random Face Trisection Graph (14999 Vertices, ~ 30000 -45000 Edges, $1/2N^{\text{th}}$ Vertex Root)	220
A.3.21	Random Face Trisection Graph (14999 Vertices, ~ 30000 -45000 Edges, $3/4N^{\text{th}}$ Vertex Root)	221
A.3.22	Random Face Trisection Graph (14999 Vertices, ~ 30000 -45000 Edges, N^{th} Vertex Root)	221
A.4.1	Mean Segment Depth for Biconnected Sub-Graphs of Triangular Prism Graph (14999 Vertices, ~ 30000 -45000 Edges, 1^{st} Vertex Root)	222
A.4.2	Mean Segment Depth for Biconnected Sub-Graphs of Triangular Prism Graph (14999 Vertices, ~ 30000 -45000 Edges, $1/2N^{\text{th}}$ Vertex Root)	222
A.4.3	Mean Segment Depth for Biconnected Sub-Graphs of Triangular Prism Graph (14999 Vertices, ~ 30000 -45000 Edges, $3/4N^{\text{th}}$ Vertex Root)	223
A.4.4	Mean Segment Depth for Biconnected Sub-Graphs of Triangular Prism Graph (14999 Vertices, ~ 30000 -45000 Edges, N^{th} Vertex Root)	223
A.4.5	Mean Segment Depth for Biconnected Sub-Graphs of Planar Wheel Graph (14999 Vertices, ~ 30000 -45000 Edges, 1^{st} Vertex Root)	224
A.4.6	Mean Segment Depth for Biconnected Sub-Graphs of Planar Wheel Graph (14999 Vertices, ~ 30000 -45000 Edges, $1/4N^{\text{th}}$ Vertex Root)	224
A.4.7	Mean Segment Depth for Biconnected Sub-Graphs of Planar Wheel Graph (14999 Vertices, ~ 30000 -45000 Edges, $1/2N^{\text{th}}$ Vertex Root)	225
A.4.8	Mean Segment Depth for Biconnected Sub-Graphs of Planar Wheel Graph (14999 Vertices, ~ 30000 -45000 Edges, $3/4N^{\text{th}}$ Vertex Root)	225
A.4.9	Mean Segment Depth for Biconnected Sub-Graphs of Planar Wheel Graph (14999 Vertices, ~ 30000 -45000 Edges, N^{th} Vertex Root)	226
A.4.10	Mean Segment Depth for Biconnected Sub-Graphs of Ordered Face Trisection Graph (14999 Vertices, ~ 30000 -45000 Edges, 1^{st} Vertex Root)	226
A.4.11	Mean Segment Depth for Biconnected Sub-Graphs of Ordered Face Trisection Graph (14999 Vertices, ~ 30000 -45000 Edges, $1/4N^{\text{th}}$ Vertex Root)	227
A.4.12	Mean Segment Depth for Biconnected Sub-Graphs of Ordered Face Trisection Graph (14999 Vertices, ~ 30000 -45000 Edges, $1/2N^{\text{th}}$ Vertex Root)	227
A.4.13	Mean Segment Depth for Biconnected Sub-Graphs of Ordered Face Trisection Graph (14999 Vertices, ~ 30000 -45000 Edges, $3/4N^{\text{th}}$ Vertex Root)	228
A.4.14	Mean Segment Depth for Biconnected Sub-Graphs of Ordered Face Trisection Graph (14999 Vertices, ~ 30000 -45000 Edges, N^{th} Vertex Root)	228
A.4.15	Mean Segment Depth for Biconnected Sub-Graphs of Random Face Trisection Graph (14999 Vertices, ~ 30000 -45000 Edges, 1^{st} Vertex Root)	229

A.4.16	Mean Segment Depth for Biconnected Sub-Graphs of Random Face Trisection Graph (14999 Vertices, ~30000-45000 Edges, $1/4N^{\text{th}}$ Vertex Root)	229
A.4.17	Mean Segment Depth for Biconnected Sub-Graphs of Random Face Trisection Graph (14999 Vertices, ~30000-45000 Edges, $1/2N^{\text{th}}$ Vertex Root)	230
A.4.18	Mean Segment Depth for Biconnected Sub-Graphs of Random Face Trisection Graph (14999 Vertices, ~30000-45000 Edges, $3/4N^{\text{th}}$ Vertex Root)	230
A.4.19	Mean Segment Depth for Biconnected Sub-Graphs of Random Face Trisection Graph (14999 Vertices, ~30000-45000 Edges, N^{th} Vertex Root)	231
A.5.1	Permutable, Flippable Graph (1002-15002 Vertices, 1^{st} Vertex Root)	231
A.5.2	Permutable, Flippable Graph (1002-15002 Vertices, $1/4N^{\text{th}}$ Vertex Root) . . .	232
A.5.3	Permutable, Flippable Graph (1002-15002 Vertices, $1/2N^{\text{th}}$ Vertex Root) . . .	232
A.5.4	Permutable, Flippable Graph (1002-15002 Vertices, $3/4N^{\text{th}}$ Vertex Root) . . .	233
A.5.5	Permutable, Flippable Graph (1002-15002 Vertices, N^{th} Vertex Root)	233

Abstract

The first linear-time planarity testing algorithm was developed in 1974 by Hopcroft and Tarjan (H&T) [32] using a method to split a biconnected graph up into edge disjoint paths and then sequentially embed them to test for planarity (a *path addition* method). Shortly afterwards Booth and Leucker [5] developed an alternative *vertex addition* linear-time planarity test, based on the earlier work of Lempel, Evan and Cederbaum [47], using a new PQ-Tree data structure. Since then there have been many developments in PQ-Tree *vertex addition* (and related PC-Tree *edge addition*) methods including authors such as: Chiba et al. [14]; Shih & Hsu [35, 69]; Boyer and Myrvold [10, 11]; and Haeupler and Tarjan [29].

In comparison, *path addition* has changed very little from the original algorithm. In 1984, Williamson [84] showed how H&T's algorithm can be extended to find Kuratowski sub-graphs in the event of a non-planar graph; and, in 1993, Mehlhorn, Mutzel and Näher [53] produced an implementation (in C) of H&T's algorithm and extended it to create a planar embedding of a graph. This has remained the state-of-the-art in *path addition* algorithms for over a decade. Recently¹, de Fraysseix formulated an algorithm [15, 17], based on Trémaux Trees and a characterisation of planarity by W. Wu [87]; this may prove to be a highly optimised version of H&T's algorithm but is difficult to definitively prove as only an outline of its planarity testing phase is provided. These authors represent the majority of the work on *path addition* methods of planarity testing and embedding; indicating that it receives little attention compared to *vertex* or *edge addition* methods.

This thesis attempts to reinvigorate the field of *path addition* and demonstrates:

- How Trémaux Trees, which allow undirected connected graphs to be represented as a simple partial order relationship, are fundamentally related to H&T's planarity testing algorithm and includes some related invariant properties of these trees;
- That the restriction on H&T's planarity testing algorithm to test undirected biconnected graphs can be relaxed to undirected connected graphs;
- How to generate all possible embeddings of a biconnected component and how to extend this to generate all possible embeddings of separable graphs; and
- Empirical Testing of various graph types and sizes to validate these results.

¹ At approximately the same time as the first edition of this work

Acknowledgements

I would like to thank the various people who have made this thesis possible:

First (and foremost) Angela, my wife, for her support and love; looking back, she has been there throughout sharing in the highs (amongst many other memories: the Rutherford lunches; taking me to the Opera (?); getting married; and two wonderful sons) and lows (of which, I'll only mention the “three houses in three months” saga after our work relocated). It would have been improbable to do this without you.

My supervisor, Dr Peter Rodgers, for all his guidance and allowing me to chase down rabbit holes. Dr Stefan Kahrs for pointing out that `java.util.LinkedList.addAll()` is implemented to be $O(N)$ (and not $O(1)$ as expected). Professor Sally Fincher for her support when life outside academia got in the way.

And, especially, my parents, Megan & Neile, for their support throughout.

The many other friends who, although they did not directly contribute to this work, made its experience that much richer, including:

The various residents of #pie IRC channel (particularly Dr Andrew “clydefrog” Secker, Dr Jonathan “stotty” Stott, Akiko Uriu, Dr Paul “pandrews” Andrews and Dr Fred Barnes plus Monty, the often humorous bot) for their friendship, many laughs, lunches, pancakes and Pimms.

The many entertaining days/evenings courtesy of: Martin & Heather, Eliot, Jonathan and Roy; Owen, Wes, Barry, Vicky, Lara, Dan, Sam, Ben and everyone else who joined in at Reed Avenue; the members of the FNFF Club including Paul & Heather, Dave & Kim, Jan & Jenny, James, Stu and particularly, our hosts, John, Michelle & Sam; and, after departing Canterbury, Jack, Ivan, Richard and Huw.

Chapter 1

Introduction

Graph Theory is a branch of mathematics and computer science that studies graphs. Graphs can be considered: in the abstract, as mathematical structures representing relationships (edges) between pairs of objects (vertices); or topologically, as an embedding consisting of curves (edges) connecting points (vertices) on a surface. Planar Graphs are a sub-set of graphs which have an embedding in the $\mathbb{R}^2$ plane (or any other genus 0 surface) such that: each vertex is at a distinct point on that plane; each edge is a simple curve; and no edges intersect except at their common vertices. Planar Graphs are useful in a wide variety of fields including:

- Electronic circuit board design, where circuits printed onto a single layer must be planar;
- Representation of transportation networks;
- Determining isomorphism of chemical compounds (becomes easier if the graph representation is planar);
- Telecommunications networks using a spanning tree representation; and
- Graphical representations of data – edge crossings have been identified as a major cause of confusion when comprehending the relationships within a graph [61–63, 80]; therefore, determining if a graph is planar and, hence, can be embedded without edge crossing can be used to eliminate one cause of miscomprehension and improve the usability of the graph visualisation.

1.1 Motivation

One of the remaining problems in the field of planar graphs is the creation of an algorithm to identify if a graph has more than one planar embedding and to generate those embeddings.

Since 1930, when Kuratowski [44] published the first characterisation of planar graphs, there has been much research in the field of planar graphs. In 1974, Hopcroft and Tarjan (H&T) [32] published the first linear-time planarity testing algorithm and then, in 1976, Booth and Lueker (B&L) [5] produced a second such algorithm. In 1985, Chiba et al. [14] extended B&L's algorithm to generate an embedding and the same was done for H&T's algorithm in 1993 by Mehlhorn, Mutzel, and Näher; however, both these extended algorithms only generate a single embedding for a graph.

The problem is to efficiently identify if an abstract graph can be displayed in a planar fashion and once this has been achieved to identify different possible planar representations of the same graph. There has been much research in this field and, particularly over the past 34 years since the first linear time algorithm by Hopcroft and Tarjan [32], there has been much work on simplifying and improving the efficiency of the algorithms. However, there are no algorithms published¹ that can identify all planar representations of a graph in linear time.

The motivation for this thesis is to:

- Give an overview of the developments in planar graph embedding, highlighting the lack of existing algorithms to generate multiple embeddings of a graph; and
- Show that the algorithm presented is a new and novel method for generating all possible planar embeddings of a graph.

¹ Haeupler and Tarjan state a possible methodology of identifying all planar representations of a graph in their extended abstract (26th March 2008) [29] using a vertex addition method but have not provided details of the algorithm.

1.2 Thesis Aims

1. To give an explanation of Hopcroft and Tarjan's planarity test in terms of chains of related vertices and edges within a Trémaux Tree (Chapters 3-4);
2. To show that Hopcroft and Tarjan's restriction on their planarity test of only processing biconnected graphs can be relaxed to consider connected graphs, in $O(\mathcal{E})$ time complexity (Chapter 4);
3. To demonstrate the conditions where a graph allows for permutations of embeddings (Chapter 4);
4. To produce an algorithm that identifies these permutations and can iterate through all possible permutations of a biconnected graph in $O(\mathcal{P})$ time, where $\mathcal{P}$ is the total number of permutations of embeddings of the graph (Chapter 5);
5. To show how to generate an embedding (cyclic edge order) from this result in linear time (Chapter 4);
6. To demonstrate how this can be extended (without specifying a time bound) to include separable graphs; and
7. To empirically test the algorithm to demonstrate its linearity when testing for planarity (Chapter 7).

1.3 Thesis Outline

This thesis will be split into four main sections:

- Firstly, an overview to history of planar graph drawing (Chapter 2) and a commentary on the current state-of-the-art algorithms will be presented. The aim of this section is to identify the limits of current planarity testing algorithms and will show that the algorithm presented in the later sections provides a new and novel technique when compared to the current state-of-the-art.
- The second section presents the concept of a Trémaux Tree (Chapter 3), a tree structure which defines a partial order on the graph, and shows how this tree can be partitioned into edge disjoint segments (chains of elements related within the Trémaux Tree) which can be used to test for planarity (Chapter 4); additionally, the partial order is investigated to show how it can be used to identify multiple embeddings.
- The third section relates the previously defined concepts to an algorithm (Chapter 5), implementing the planarity test (and identification of permutations within an embedding) in linear time. Additionally, it will show that, given a specific permutation of those segments that can be permuted that an embedding can be generated in linear time.
- The final section will present an experimental analysis (Chapter 7) of the algorithm to further demonstrate the algorithm's linear run time and will draw the conclusions from the previous sections together to show the overall efficiency of the algorithm.

Chapter 2

A History of Planarity Testing and Embedding

The use of graph drawing (the visual layout of graphs) as a means to describe relationships has been around for millennium. It has only been in the last 350 years that graph theory (using mathematical abstractions to solve graph problems) has begun to introduce mathematical processes into the field.

The earliest explicit forms [42, 43] of graph drawing were probably Nine Men's Morris games [56] found on roof slabs in a temple began by Rameses I (1400-1366 B.C.). Other examples of early graphs include depictions of family trees where descriptions of such trees occur as far back as the Roman era but the earliest surviving examples date from the 11th Century.

The transition between early and modern graph drawing was probably marked by Leibniz's (1646 - 1716) new kind of geometry "Geometria Situs" (later to be called topology). In his letter of September 8th 1679 to Huygens, he describes it as a "*new characteristic, completely different from algebra, which will have the great advantage to represent to the mind exactly and naturally, though without figures, everything which depends on the imaginations*" [46].

The planarity problem is to find out if for a given undirected graph, a graph can be drawn in the plane without any edge crossings. The planarity problem was characterised by Kuratowski [44], in 1930, when he proved that every non-planar graph contains a sub-graph that upon removal of degree two vertices is isomorphic to a complete graph on five vertices (K_5) or to a complete bipartite graph on six vertices ($K_{3,3}$). Conversely, no planar graphs contain either of these graphs. Although this gives a simple solution to the planarity problem, testing these conditions is of no practical value as Hopcroft and Tarjan [32] (H&T) state an algorithm designed to find these graphs would be $O(V^6)$.

Instead of looking for the Kuratowski graphs, the current best approach is to attempt to construct a representation of a planar embedding of the graph, building the graph up incrementally until either the graph is completely embedded or a non-planar element is identified. The first such algorithm was proposed by Auslander and Parter [2] in 1961. The algorithm removed a cycle from the graph causing it to fall to pieces and then starting from the cycle, pieces were reattached recursively until the graph was rebuilt (planar) or a piece was obstructed (non-planar). Unfortunately the algorithm contained an error allowing an indefinite loop and Goldstein [26] later (1963) correctly formulated algorithm, using iteration instead of recursion. Later implementations of this algorithm have been shown to have $O(V^3)$ time-complexity, where V is the number of vertices in the graph.

Alternate methods have been presented including Lempel, Even and Cederbaum's [47] (1966) algorithm for vertex addition, which builds the graph from a single vertex by adding incident edges and repeating using careful ordering of the vertices. Although they gave no implementation or time-bound for the algorithm it has been shown that it can be implemented in $O(V^2)$ time. Mondshien [55] also proposed an $O(V^2)$ algorithm for adding a single vertex at a time until the entire graph was constructed, again relying on careful ordering of the vertices.

H&T [31], using depth-first search and a complicated program devised a variant of Goldstein's algorithm to run in $O(V \cdot \log(V))$ and subsequently discovered an improved algorithm with $O(V)$ time complexity, which appears in Tarjan's [72] dissertation. H&T [32] further simplified their algorithm in 1974, producing what is acknowledged as the first linear-time planarity test.

A second linear algorithm developed by Booth and Lueker (B&L) [5], using PQ-trees, in 1976 was the next seminal work in the field and these two have formed the basis on which most further $O(V)$ algorithms have built. Both, H&T and B&L's approaches have had subsequent papers to simplify, extend or improve those algorithms, including:

- Williamson [84, 85] and Mehlhorn, Mutzel, and Näher's [52–54] extensions to H&T's algorithm to generate Kuratowski sub-graphs and a planar embedding (respectively);
- Kocay's [41] graph theoretical analysis of H&T algorithm; and
- Work extending/improving B&L's algorithm such as:
 - Chiba et al's [14] extension to generate an planar embedding;
 - A method using PC-Trees, initially proposed, by Shih and Hsu [68] (although not addressing important implementation issues [29, pg. 2]) and subsequent, incorrectly formulated, revisions in [33–35, 69] before a correct algorithm was presented, and later expounded upon, by Boyer et al. [7, 12]; and
 - A similar algorithm by Boyer et al [9–11, 13] also using PC-Trees; and
 - A more recent PQ-tree algorithm by Haeupler and Tarjan [29] which, although only outlined, asserts a method of using PQ-trees in a similar fashion to PC-trees and of representing all possible embeddings of a biconnected graph.

A more detailed investigation of the most prominent work will be pursued below.

2.1 Euler Characteristic – 1752

Leonard Euler (15th April 1707 - 18th September 1783) [22] is one of the pre-eminent and prolific Mathematicians of the 18th century. In 1736, he solved the Seven Bridges of Königsberg problem by proving that there was no Eulerian Circuit when trying to walk around the city of Königsberg by only crossing each bridge once. He generalised the problem by proving that there would be an Eulerian Circuit (a path visiting each vertex only once) for a graph if all the vertices, except the start and end points, have an even degree. This is considered to be the first theorem of graph theory and, more specifically, the first planar graph theorem. Euler's work, in 1752, developing the Euler Characteristic and the Euler Formula became a foundation stone in the field of topology.

For a finite connected planar graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ (with set $\mathcal{V}$ of vertices and set $\mathcal{E}$ of edges) and a corresponding planar embedding (with set $\mathcal{F}$ of faces) on a surface (with genus¹ 0) then Euler's Formula (alternately known as Euler's Polyhedral Formula) defines the relationship between the number of vertices, edges and faces such that:

$$|\mathcal{V}| - |\mathcal{E}| + |\mathcal{F}| = 2 \quad (2.1.1)$$

Poincaré [59] generalised Euler's polyhedral formula for other surfaces such that, given a surface with genus g then:

$$|\mathcal{V}| - |\mathcal{E}| + |\mathcal{F}| = \chi(g) \quad (2.1.2)$$

Where $\chi(g)$ is the Euler Characteristic of that surface and is given by:

$$\begin{aligned} \chi(g) &= 2 - 2g && \text{(For an orientable surface; i.e. the surface of a sphere or torus.)} \\ \chi(g) &= 2 - g && \text{(For an non-orientable surface; i.e. the surface of a Klein bottle.)} \end{aligned} \quad (2.1.3)$$

For the remainder of this work, it shall be assumed that any planar embedding is in an orientable plane of genus 0 ($\mathbb{R}^2$ plane or the surface of a sphere) and Euler's Formula (rather than Poincaré's generalisation) applies.

Euler's Formula gives rise to one of the basic tests for non-planarity: for a maximal planar² graph then all the faces are bounded by three edges (triangular) and each edge connects to two faces; hence, $3|\mathcal{F}| = 2|\mathcal{E}|$. Therefore, any simple planar graph, with $|\mathcal{V}| \geq 3$, then the upper bound for the number of edges $|\mathcal{E}|$ is:

$$|\mathcal{E}| \leq 3|\mathcal{V}| - 6 \quad (2.1.4)$$

Therefore any simple planar graph exceeding this limit is non-planar.

¹ The genus of a surface is, roughly, equal to the number of holes in that surface; i.e. the infinite two-dimensional plane, $\mathbb{R}^2$, or the surface of a sphere have genus 0 and the surface of a torus has genus 1

² A maximal planar graph is a simple graph with a planar embedding having the additional characteristic that no edge can be added to the graph without making the graph either non-simple or non-planar.

2.2 Trémaux's (Depth-First Search) Algorithm – 1882

In 1882 Lucas published *Récréations Mathématiques* [51] containing an algorithm by Monsieur Trémaux to find the solution for any maze (or investigate all passageways and junctions then return to the maze's entrance) without traversing any passageway more than twice. Trémaux's algorithm is (and, may be, the first published example of) a Depth-First Search (DFS) algorithm. A summary of Trémaux's algorithm is given below:

- 1 Given a maze (graph) composed of a set of passageways (edges) terminating at junctions (vertices) or dead-ends (vertices with only a single adjacent edge) then progress through the maze can be tracked by marking the ends of each passageway, using the markers:
 - 2 • “In” (denoting a journey inbound along that passage to that junction);
 - 3 • “Out” (denoting an outbound journey along that passage); and/or
 - 4 • “First” (denoting the passageway first used to reach that junction).
- 6 Trémaux uses the marking system, on passage entrances, such that: “First” is denoted by a single line (and all other markers are ignored); “Out” is denoted by a cross (two lines); and “In” is not given a notation as that entrance is, immediately, additionally marked as either “First” or “Out”.
- 8 On arriving at a junction along a passageway: mark the exit to that passage with an “In” marker; and if the exit to the passage travelled is otherwise unmarked, and all other passages from that junction are also unmarked, then additionally mark the passage with a “First” marker.
- 10 On departing from a junction, mark the entrance to the chosen passageway with an “Out” marker. The passage chosen to depart from a junction is, in order of preference (passages marked with “Out” are never followed a second time):
 - 11 • A passageway marked, solely, with “In”;
 - 12 • A randomly selected unmarked passageway;
 - 13 • The passageway marked as “First”.

Unfortunately, while the algorithm was correctly formulated, the proof provided by Lucas was deficient [3].

2.2.1 Tarry's Algorithm – 1895

In 1895, Tarry [74][3, English Translation] published an algorithm is based on the similar principle:

"Do not return along the passage which has led to a junction for the first time unless you cannot do otherwise."

Tarry is sometimes quoted [15] as having republished Trémaux's algorithm, however, this is not the case. The differences from Trémaux's algorithm are summarised below:

- 6 Tarry uses the marking system, on passage entrances, such that: “First” is denoted by a three lines (and “In” and “Out” markers are ignored); “Out” is denoted by a two lines (and any “In” marker is ignored); and entrances marked, solely, with “In” are denoted by a single line.
- 10 On departing from a junction, mark the entrance to the chosen passageway with an “Out” marker. The passage chosen to depart from a junction is, in order of preference (passages marked with “Out” are never followed a second time):
 - 11 • A randomly selected passageway from those passages that are either unmarked or marked, solely, with “In”;
 - 12 •
 - 13 • The passageway marked as “First”.

Both algorithms find solutions to any maze by traversing each edge at most twice; however, Tarry's algorithm prefers to follow paths randomly selected from the set of unmarked paths and paths marked, solely, with "In" whereas Trémaux's algorithm prefers to follow paths marked, solely, with "In" and only if no such paths exists then to follow a randomly selected unmarked path. Since, if paths marked "In" are chosen prior to unmarked paths then, Tarry's algorithm can generate the same result as Trémaux's; therefore Trémaux's algorithm is a specific sub-class of Tarry's algorithm and proof Tarry's algorithm solves any maze (as given by Tarry [74]) also holds for Trémaux's algorithm.

2.2.2 Trémaux Trees

The edges of a graph can be partitioned into the sets of tree, cotree and cross edges such that:

- The set of Tree edges forms a spanning tree of the graph. A partial order relationship can be defined over the set of vertices such that vertex u is defined to be related to vertex v (as an ancestor or equal to) if there is a path of Tree edges containing u between v and a given vertex rooting the spanning tree (typically defined to be the vertex first processed by the algorithm generating the spanning tree). In terms of Trémaux's, or Tarry's, algorithm then each Tree edge has one end marked "First" (and the vertex at the end not marked "First" is an ancestor of the vertex at the end marked "First").
- The set of Cotree edges are those non-Tree edges (where neither edge is marked "First") such that the terminating vertices of each Cotree edge are related; and
- The set of Cross edges are those non-Tree edges (where neither end is marked "First") such that the terminating vertices of each Cross edge are not related (instead those vertices are in divergent branches of the rooted spanning tree).

A rooted spanning tree of a connected graph is defined to be a Trémaux Tree (alternately known as a Depth-First Search Tree) if, and only if, all non-Tree edges of that graph are Cotree edges; hence, for a Trémaux Tree, the terminating vertices of every edge are related since all edges are either Tree or Cotree edges).

Bondy and Murty [4, pg. 141-142], amongst other graduate text-books, show³ (using two time functions corresponding to the time vertices are first and last visited) that given a spanning tree generated by traversing a connected graph using a DFS (Trémaux's) algorithm then all vertices terminating edges of that graph share an ancestor-descendant relationship; hence, the spanning tree is a Trémaux Tree.

In contrast, Tarry's algorithm can generate all three types of edges whilst, as an aside, (and assuming the additional constraint of a graph without looping edges³) a Breadth-First Search (BFS) algorithm generates only Tree and Cross edges.

Trémaux (DFS) Trees (and BFS-trees) are used extensively within many graph related algorithms, a sample of which (relating to planarity testing) are discussed later in this chapter. This is due to: the invariant properties of these trees (some of which are detailed, for Trémaux Trees, in Chapter 3); and linear time and memory implementations of DFS (and BFS) algorithms.

³ Bondy and Murty's proof only considers non-looping edges; however, all looping edges are Cotree edges since: every vertex is contained within the path of Tree edges from the root vertex to itself so every vertex is related to itself; and that Trémaux's algorithm will not mark either end of a looping edge with "First" (as the looping passageway's entrance will have been marked prior to testing for unmarked passageways upon reaching the loop's exit).

2.3 Jordan Curve Theorem – 1887

The Jordan Curve Theorem establishes⁴ that, within the plane $\mathbb{R}^2$, a Jordan curve formed by two Jordan arcs A and B (which both terminate at points p and q but do not otherwise intersect) then:

1. The complement $\mathbb{R}^2 \setminus (A \cup B)$ can be partitioned into two distinct components: the (unbounded) set of exterior points; and the (bounded) set of interior points. Given a point $x \in \mathbb{R}^2 \setminus (A \cup B)$ then there exists an upward-directed ray emanating from x and the crossing parity of that ray and a Jordan arc is whether the ray crosses the arc an odd or even number of times. Considering the crossing parities for the ray emanating from any point relative to the arcs A and B then it is an exterior point if the parities are equal and an interior point if the parities are unequal;
2. Both components are non-empty (the interior can be shown to contain an incircle with non-zero radius and the exterior is unbounded);
3. All points within each component are path-connected; and
4. The pair of components are disconnected such that $A \cup B$ is their common boundary.

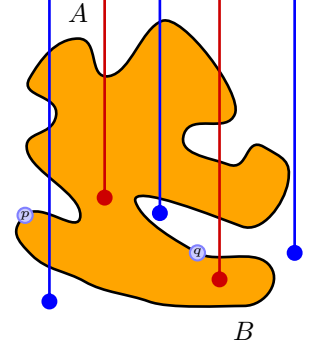


Figure 2.3.1: Rays Intersecting a Jordan Curve

Figure 2.3.1 gives a graphical representation of a Jordan curve formed by two Jordan arcs, A and B , and five points within the complement and their corresponding upwards directed rays. Each point and corresponding ray is coloured: in red, when the ray's crossing parities are unequal, representing an interior point; and in blue, when the ray's crossing parities are equal, representing an exterior point.

A corollary [30, pg. 49] to the Jordan Curve Theorem is that given Jordan arcs A , B and C , which have the same endpoints but do not otherwise intersect, then the Jordan arc A (respectively B , C) partitions the component of the Jordan Curve $B \cup C$ (respectively $A \cup C$, $A \cup B$) containing that arc into two distinct connected sub-components separated by that arc.

Hopcroft and Tarjan [32, pg. 551-552] state a further corollary: given the arcs A , B and C , as previously described, then if the clockwise orientation of the arcs about p are in the order $\langle A, B, C \rangle$ then the clockwise orientation of those arcs about q is in the order $\langle A, C, B \rangle$.

Figure 2.3.2 gives a graphical example of both of these corollaries such that A (respectively B , C) partitions the exterior (respectively interior, exterior) component of the Jordan curve $B \cup C$ (respectively $A \cup C$, $B \cup C$) into two regions and that the arcs have the theorised clockwise orientations about p and q .

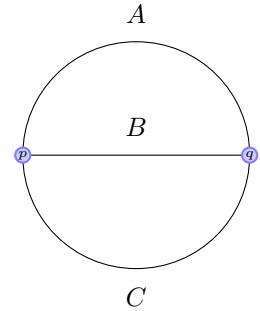


Figure 2.3.2: Three Non-Intersecting Jordan Arcs With Shared Endpoints

⁴ In the words of Jordan [39, pg. 99], “Il est donc établi que toute courbe continue C divise le plan en deux régions, l’une extérieure, l’autre intérieure, cette dernière ne pouvant se réduire à zéro, car elle contient un cercle de rayon fini.”

2.3.1 Historical Note on the Correctness of Jordan's Proof

Guggenheimer [28] and Hales [30] both provide a commentary on Jordan's proof and the subsequent proofs. However, they consider them from different perspectives: Guggenheimer gives a brief overview of the methodology used and their shortcomings; whereas Hales discusses, as a prelude to the main body of his work, the dissatisfaction within some of those papers with regards to Jordan's original proof.

Guggenheimer [28, pg. 194] notes that Jordan does not prove the theorem as he omits the proof in the simple case of polygons⁵ and that Jordan's proof of existence of two polygonal approximations to a Jordan Curve (within and without that curve) that are within a sufficiently small maximum distance from that curve (which later parts of Jordan's proof rely upon) is not convincing. However, he does note that the latter issue is filled by Young & Young in their 1906 paper "The Theory of Sets of Points" and Hales notes [30, pg. 46] the former was "*widely regarded as completely trivial*" with proofs of it appearing in undergraduate textbooks.

Hales notes similar comments from Veblen [77]⁶ (who is often referenced, see [3, pg. 141], as having produced the first correct proof of the Jordan Curve Theorem) and Osgood⁷, who notes that Jordan's proof is correct only only under the assumption of its correctness for polygons and that Jordan's proof contains assumptions. Hales goes on to quote a private correspondence from Reeken⁸ that "*Jordan's proof is essentially correct. . . Jordan's proof does not present the details in a satisfactory way. But the idea is right and with some polishing the proof would be impeccable*".

The major subject of Hales' work is not the critique of the treatment of Jordan's proof but an up-to-date version of Jordan's proof; highlighting, contrary to many [30, pg. 45], the validity of Jordan's methodology. So: whilst the first rigorous, and axiomatic, proof of the Jordan Curve Theorem can be attributed to Veblen; Jordan's original methodology can be used to prove his theorem.

⁵ "He [Jordan] accepts as obvious that a simple closed polygon separates the plane into two regions in which two points can be joined by a polygonal arc without crossing the polygon but that any polygonal arc joining two points in distinct regions must cross the polygon."

⁶ "[Jordan's proof] is unsatisfactory to many mathematicians. It assumes the theorem without proof in the important special case of a simple polygon and of the argument from that point on, one must admit at least that all details are not given"

⁷ "Es sei noch auf die Untersuchungen von C. Jordan verwiesen, wo der Satz, unter Annahme seiner Richtigkeit für Polygone, allgemein für Jordansche Kurven begründet wird. Jordan beweist hiermit mehr als die Funktionentheorie gebraucht; dagegen macht er Voraussetzungen, welche diese Theorie streng begründet wissen will"

⁸ Reeken is a co-author of the 1996 paper "A non-standard proof of the Jordan curve theorem", using similar methodology to Jordan's original proof

2.4 Kuratowski's Theorem – 1930

Kuratowski [44] gives a characterisation of non-planar graphs such that:⁹ *for a connected non-planar graph which has only a finite number of circuits, then that graph contains a sub-graph homeomorphic to a $K_{3,3}$ graph or a K_5 graph.* Conversely, no planar graph contains a sub-graph homeomorphic to either of these graphs.

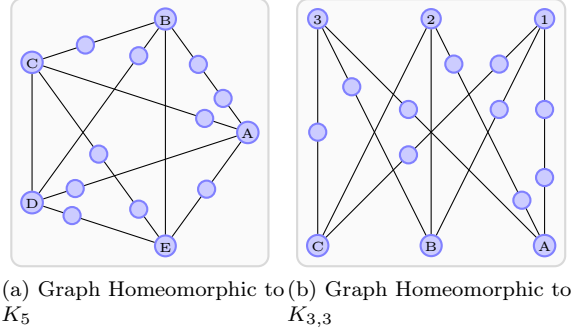


Figure 2.4.1: K_5 and $K_{3,3}$ Non-Planar Graphs

Hopcroft and Tarjan state [32, p. 550] that “[a]lthough elegant, Kuratowski’s condition is useless as a practical test of planarity; testing for such subgraphs directly may require an amount of time proportional to at least $\mathcal{V}^8$, if not much worse, where $\mathcal{V}$ is the number of vertices in the graph”. They go on to conclude that “[t]he best approach to the planarity problem seems to be an attempt to construct a representation of a planar embedding of the given graph. If such a representation can be completed then the graph is planar; if not, then the graph is nonplanar”.

Subsequent algorithms [8, 11, 17, 84] have taken this approach full circle; using the failure to construct a planar embedding to locate a sub-graph homeomorphic to K_5 or $K_{3,3}$ in linear-time. However, Hopcroft and Tarjan’s conclusion about the inefficiency of testing for non-planarity by directly searching for such sub-graphs has not (to the knowledge of the author) been refuted.

⁹ In the words of Kuratowski [44, pg. 278]: “Théorème A. A étant un continu Péanien gauche qui ne contient qu’un nombre fini de courbes simples fermées, A contient une courbe homéomorphe à la courbe de la fig. 1 [$K_{3,3}$] ou à celle de la fig. 2 [K_5].”

2.5 Hassler Whitney – 1932 & 1933

Whitney contributed much to the field of graph theory; particularly relating to graph colouring, graph isomorphism, non-separable graphs and graph duals. Of particular interest, regarding planar graphs, are two theorems: the first is a characterisation of planar graphs such that a dual exists for all planar graphs; and the second relates to 2-isomorphic graphs.

In 1932, Whitney published his paper “Non-Separable and Planar Graphs” [81] proving the theorem that:

A necessary and sufficient condition that a graph be planar is that it have a dual.

For a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ with an embedding consisting of a set $\mathcal{F}$ of faces then its dual is the graph $\mathcal{G}'(\mathcal{V}', \mathcal{E}')$ with set $\mathcal{F}'$ of faces such that there is a bijection (one-to-one correspondence) between: $\mathcal{F}$ and $\mathcal{V}'$; $\mathcal{V}$ and $\mathcal{F}'$; and $\mathcal{E}$ and $\mathcal{E}'$. A dual graph has an inverse such that $\mathcal{G}$ is the dual of $\mathcal{G}'$. Figure 2.5.1 gives an example of a graph (vertices coloured blue and edges black) overlaid by its dual (coloured red).

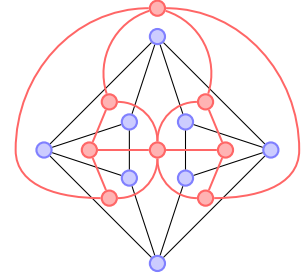


Figure 2.5.1: A Graph (blue/black) and its Dual (red)

This theorem gives an alternate and equivalent characterisation of planarity to Kuratowski’s theorem.

Of particular relevance to the subject of permutations of graph embeddings is Whitney’s 1933 publication “2-Isomorphic Graphs”; this introduces Whitney’s 2-Isomorphism Theorem which gives the conditions for two graphs to be isomorphic¹⁰ and gives rise to the operation known as a “*Whitney flip*” (which generates a permutation of the embedding of a *2-separable* planar graph).

Given a partition $(\mathcal{E}', \mathcal{E} \setminus \mathcal{E}')$ on the set $\mathcal{E}$ of edges of a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, such that both sub-graphs induced by those partitions are non-empty and connected, then this is a *k-separation* if there are exactly *k* boundary vertices incident with $\mathcal{E}'$ and $\mathcal{E} \setminus \mathcal{E}'$. Given an embedding Γ of $\mathcal{G}$ containing a *2-separation* $(\mathcal{E}', \mathcal{E} \setminus \mathcal{E}')$ then a *Whitney flip* is an operation which removes $\mathcal{E}'$ from Γ and re-embeds the mirror image (re-attaching $\mathcal{E}'$ at the two boundary vertices), generating a permutation of the embedding. The corresponding effect this operation has on the dual graph is known as a *Whitney twist*.

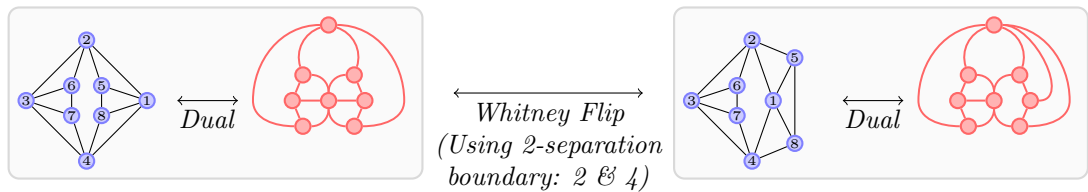


Figure 2.5.2: Example of a *Whitney Flip*

A corollary to Whitney’s 2-Isomorphism Theorem¹¹ is that any 3-connected graph has a unique embedding (since no *2-separation* of that graph exists so no *Whitney flips* can be performed, hence, there are no permutations of the embedding).

¹⁰In the words of Whitney [81, pg. 245]: “[T]wo graphs G and G' are 2-isomorphic if one can be transformed into the other by operations of the following two types: (2) The arrangement of the components in the graph is altered. (3) If $G = H_1 + H_2$, where H_1 and H_2 have just the vertices a and b in common and these vertices are connected in both H_1 and H_2 , then H_1 is turned around at these vertices.”

¹¹[81, pg. 246] “COROLLARY. If G and G' satisfy the conditions of the above theorem and one of them is triply connected, then both are, and the two graphs are strictly isomorphic.”

2.6 Lempel, Even, and Cederbaum's $O(V^2)$ Vertex Addition Planarity Test – 1967

Lempel, Even, and Cederbaum's algorithm [47] tests for planarity on a biconnected graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$. The algorithm is a two stage process:

1. An st -order (described below) is generated for the set of vertices; and
2. Planarity is tested vertex-by-vertex such that properties of the Bush Form (as described below) of the connected planar sub-graph $\mathcal{G}_k$ (induced by vertices with st -numbering $1 \dots k$) can be used to determine the existence of a planar embedding (and corresponding Bush Form) of $\mathcal{G}_{(k+1)}$.

2.6.1 st -ordering

Given a biconnected graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ with an edge $\{s, t\} \in \mathcal{E}$, in an st -ordering of $\mathcal{G}$, then: s receives number 1; t receives number $|\mathcal{V}|$; and each other vertex receives a unique number such that it is adjacent to a lower-numbered and a higher-numbered vertex. Even and Tarjan [23] show that it is always possible to find an st -numbering for a biconnected graph.

The following properties exist regarding the sub-graph $\mathcal{G}_k$ (of graph $\mathcal{G}$), induced by vertices $v_1 \dots v_k$ with corresponding st -numbers :

- $\mathcal{G}_k$ is connected, since each vertex (except v_1) connects to a lower st -numbered vertex;
- $(\mathcal{G} - \mathcal{G}_k)$ is connected, since each vertex (except $v_{|\mathcal{V}|}$) connects to a higher st -numbered vertex; and
- Given the previous two properties and a planar embedding Γ of $\mathcal{G}$ then considering the sub-embeddings Γ_k and $(\Gamma - \Gamma_k)$ of $\mathcal{G}_k$ and $(\mathcal{G} - \mathcal{G}_k)$, respectively, then all elements of $(\mathcal{G} - \mathcal{G}_k)$ must be contained within a single face of Γ_k (otherwise there is an edge crossing, contradicting the fact that the embedding is planar) which can be designated as the external face. It follows that all vertices (including v_k) in $\mathcal{G}_k$ that are adjacent to vertices in $(\mathcal{G} - \mathcal{G}_k)$ must be on the boundary of this external face.

2.6.2 Bush Form

This final property of st -orderings gives rise to a visualisation, known as a Bush Form ($\mathcal{B}_k$), of the planar embedding of $\mathcal{G}_k$ composed of:

- Γ_k (oriented such that $(\Gamma - \Gamma_k)$ could be contained in the external face);
- All edges in $(\mathcal{G} - \mathcal{G}_k)$ connected to a vertex in $\mathcal{G}_k$ (designated as virtual, or half, edges), embedded in the external face; and
- A virtual vertex for each disconnected end of a virtual edge (assigned the st -number corresponding to the vertex in $(\mathcal{G} - \mathcal{G}_k)$ that edge would connect to).

The edges of the Bush Form can be given directionality outbound from (inbound to) the lower (higher) st -numbered connected vertex. Commonly, Bush Forms are visualised (see Figure 2.6.1) such that virtual vertices are positioned in a horizontal line above, with virtual edges ascending from, the embedding of $\mathcal{G}_k$.

Figure 2.6.1 shows a sample st -ordered planar graph and examples of the corresponding Bush Forms for successive sub-graphs.

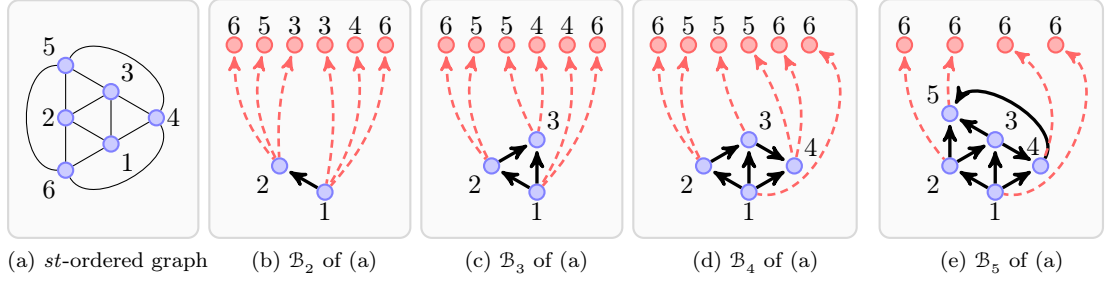


Figure 2.6.1: Sample Planar Graph and corresponding Bush Forms $\mathcal{B}_2 \dots \mathcal{B}_5$

The edges of a graph can be partitioned into distinct pieces such that each piece is a bridge edge or a maximal biconnected component; these pieces form a tree (since any cycle involving multiple pieces means those pieces were not maximal). Within the Bush Form: the embedding of each biconnected piece can be flipped, reversing the cyclic order of vertices on the external face of that piece; and the cyclic order of the embedding of pieces outbound from a given vertex can be permuted. Using this, the cyclic order of the virtual edges (and vertices) of a Bush Form can be changed via a sequence of flips or permutations of the pieces.

Lempel, Even, and Cederbaum [47] show that given the (planar) Bush Form $\mathcal{B}_k$ then $\mathcal{B}_{k+1}$ can be formed (and is planar) if, and only if, there is a sequence of flips or permutations of the embedded pieces of $\mathcal{B}_k$ such that all virtual vertices labelled v_{k+1} are consecutive within the cyclic order of virtual vertices.

This process, of generating an st -order then using the inductive loop of reordering the embedding and coalescing consecutive virtual vertices to append successive st -ordered vertices in a planar manner to the embedding, forms the basis for Lempel, Even, and Cederbaum's algorithm. A non-planar graph is found when an inductive step is reached where it is impossible to reorientate the Bush Form such that the virtual vertices with the next st -number are ordered consecutively within the cyclic order of virtual vertices.

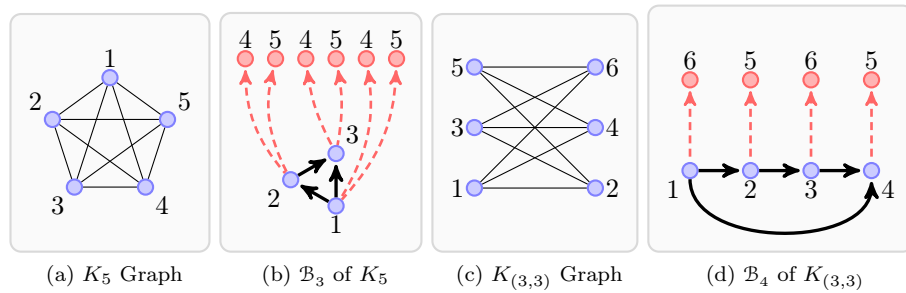


Figure 2.6.2: Non-Planar Graphs K_5 and $K_{(3,3)}$ and their corresponding Bush Forms.

Figure 2.6.2 gives an example of K_5 and $K_{3,3}$ and their Bush Forms. The Bush Form $\mathcal{B}_3$ of K_5 has three permutable pairs of virtual vertices labelled 4 and 5 around a biconnected piece; at most, only two of the three virtual vertices labelled 4 can be consecutive within the cyclic order and therefore the inductive step to generate $\mathcal{B}_4$ fails. Similarly, the Bush Form $\mathcal{B}_4$ of $K_{3,3}$ contains four virtual vertices each connected to different vertex on a biconnected piece such that the st -numbering of those virtual vertices alternates; therefore the only reordering possible is flipping the biconnected piece and, again, the inductive step fails. Therefore, in both cases, the graphs are declared non-planar.

2.7 Wagner's Conjecture (Theorem) - 1970

An undirected graph $\mathcal{H}$ is a minor of the graph $\mathcal{G}$ if there exists a graph $\mathcal{G}'$, isomorphic to $\mathcal{H}$, which can be formed from the transformation of $\mathcal{G}$ through zero or more edge deletions and/or edge contractions (the deletion of an edge whilst simultaneously coalescing its two incident vertices to form a single replacement vertex).

Wagner's conjecture [79] (now Wagner's Theorem [65, 66]) is a characterisation of planarity relating to forbidden minors and asserts that *the family of planar graphs does not contain any graph which has K_5 or $K_{3,3}$ as a minor*.

The Robertson-Seymour Theorem [66] is a generalisation, and proof¹², of Wagner's conjecture; it asserts that every minor-closed¹³ family $\mathcal{F}$ of graphs can be defined by a finite set of forbidden minors (graphs that are not minors of any graph in $\mathcal{F}$).

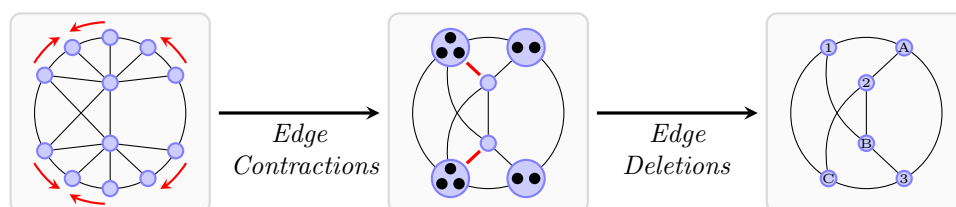


Figure 2.7.1: Example of a $K_{3,3}$ Graph Minor

¹²Proved via a series of 20 papers culminating with [66].

¹³A minor-closed family $\mathcal{F}$ of graphs is such that for every graph in $\mathcal{F}$ then any minor of those graphs is also in $\mathcal{F}$.

2.8 Hopcroft and Tarjan's Path Addition Planarity Test – 1974

Hopcroft and Tarjan's (H&T) algorithm developed from an earlier algorithm proposed by Auslander and Parter [2] (whose algorithm contained an error inducing an infinite loop and a correct implementation was provided by Goldstein [26]). In 1971, Hopcroft and Tarjan [31] formulated a $O(V \log(V))$ time bounded variant of Goldstein's algorithm which they later improved to $O(V)$ in Tarjan's dissertation [72] and simplified in their 1974 paper "Efficient Planarity Testing" [32] (although corrections to their algorithm were later published by Deo [21] in 1976).

Of note, to this time line, is the (independently derived) 1964 paper of Demoucron, Malgrange, and Petruiset [20]; who presented an algorithm with a similar premise (of separating the graph into a cycle and forest before recursively separating each tree of the forest into sub-forests) as Auslander and Parter's but, alternatively, tests planarity by locating faces which can accept a valid embedding.

The H&T algorithm takes a biconnected graph (however Tarjan [73] gives a linear-time algorithm for separating a connected graph into its biconnected components) and processes it using a three phase process:

In the first phase, a Depth-First Search (DFS) is performed over the graph generating a *palm* tree (alternatively known as a DFS-tree or a Trémaux Tree) and assigning: each vertex a DFS-index; and categorises each edge as either a *tree* or a *frond* edge (also known as a back or cotree edge) and gives it a direction. During backtracking, the algorithm calculates, for each vertex, the lowest, and second lowest, indexed vertices reachable using the descendant portion of the *palm* tree. These pieces of information are then used: to bucket sort the edges; create a adjacency list of all edges outbound from each vertex; and, within the last phase, to compare paths generated in the second phase to test for non-planarity.

The second phase involves a second DFS, considering only the (now directed and sorted) outbound edges from each vertex; first identifying a cycle within the (biconnected) graph then, as the DFS backtracks and follows new branches, identifies successive paths. The result is that the graph is partitioned into edge-disjoint paths.

The final phase takes these paths, in order of generation, and considers their embedding. The first path is a cycle which separates the embedding into two faces. Paths branching from that cycle connect to it at two distinct points and, if planar, can then be embedded on one side of that cycle or the other; those paths then define their own cycle and paths branching off that new path can then be recursively considered relative to the cycle it forms. As each successive path is considered for embedding, if it overlaps a previously embedded segment then it must be on an opposite side of the cycle it branches from compared to that previous segment.

H&T use two stacks, designated as *left* and *right*, to represent the sides of a separating cycle upon which paths can be embedded and a third stack of *blocks* of paths. A block represents a group of paths on the *left* and *right* stacks where the positioning of any one of those paths on the *left* or *right* stack determines the position of all other paths in that block to be on the same or the opposite stack; moving one path of a block from the *left* to *right* (or vice versa) swaps the stacks for all paths. Using this, the problem of embedding the graph is reduced to a sequence of simple tests and stack manipulations such that:

- Each successive path is considered against the blocks on the stack such that if a block determines the position of that new path (i.e. the last path of that block on the *left* or

right stack overlaps with the new path) then the last block is removed from the *blocks* stack: if both the *left* and *right* paths overlap then the graph is non-planar; if the *left* path overlaps then the paths of that block are swapped between *left* and *right* stacks and the previous block is considered; and if the *left* path does not overlap then the position of the paths on the stack remains unchanged and the previous block is considered. This process loops until either a block is found that does not overlap on either side or a non-planar case is found. If no non-planar state is reached then the new path is added to the *left* stack and a new block is formed of this new path and all paths in the deleted blocks.

- If a state is reached where the path at the top of a stack has no influence on the embedding of the next path then it can be removed from the stack, and block (which if that was the last path in the block then the block can also be removed from the stack of blocks), and the embedding can be considered against the previous path on that stack;
- An additional process is required when the algorithm backtracks to a point where all the paths separated by a cycle (as defined by the embedding of each path) are processed. At this point the blocks containing paths branching from this cycle are considered, having removed paths from blocks which have no influence on the embedding, and if a block contains paths on both the *left* and *right* stacks (Figure 2.8.1) then the graph is non-planar; if there are only paths on the *left* stack (the case never arises where there are just paths on the *right* stack) then those blocks are merged with the previous block since subsequent paths may overlap.

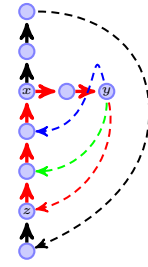


Figure 2.8.1: Paths (Green and Blue) on *Left* and *Right* of Cycle (Red) Formed by Path $(x \rightarrow y \rightarrow z)$.

2.8.1 Example of H&T's Algorithm

Figure 2.8.2 is an example graph (taken from [32, pg. 554]) that shows the order in which the depth-first search, of the first phase of H&T's algorithm, visits vertices (ascending alphabetical order) and edges (ascending numerical order). The paths formed during the second phase are shown in figure 2.8.3 and are numbered in ascending numerical order of the order they are generated.

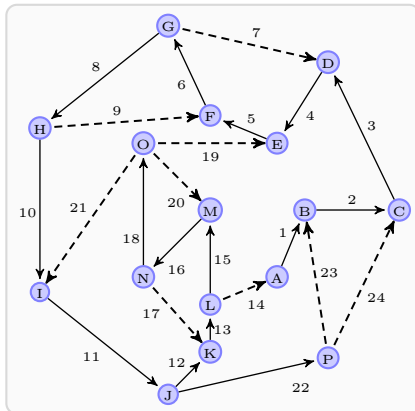


Figure 2.8.2: Example Graph and Sample Path Taken by DFS

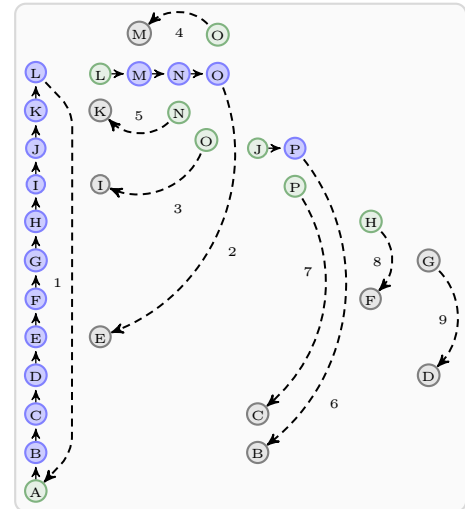


Figure 2.8.3: Disjoint Paths Generated by Second Phase of H&T's Planarity Test

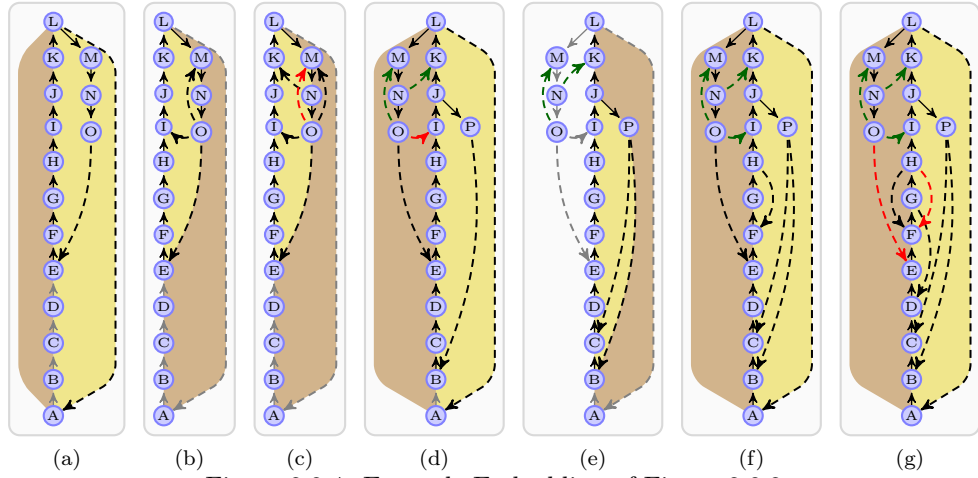


Figure 2.8.4: Example Embedding of Figure 2.8.2

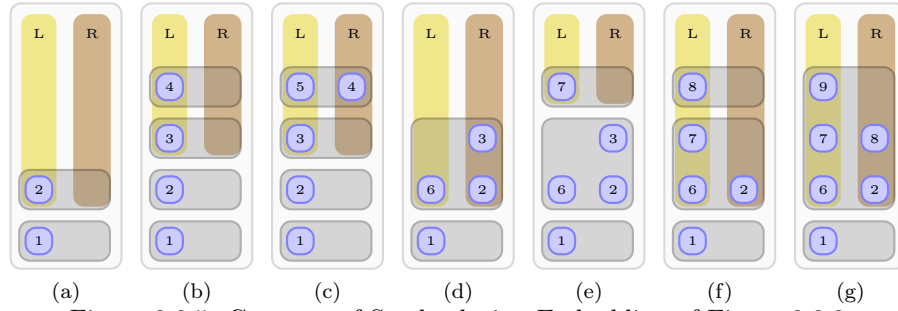


Figure 2.8.5: Contents of Stacks during Embedding of Figure 2.8.2

Figure 2.8.4 shows the process of embedding the paths and figure 2.8.5 shows the corresponding state of the stacks with two columns representing the *left* and *right* stacks (showing the numbered paths from figure 2.8.3 rather than the path's end vertices as in [32, pg. 562]) and horizontal groups representing the blocks.

- (a) embeds the first (cyclic) path (adding it to the *left* stack) defining an inside (tan) and outside (brown) face. The second path is then embedded (adding it to the *left* stack and, since it does not overlap a previous path, a new block).
- The cycle formed by this second path defines two faces (tan and brown shown in (b)) and within that the third and fourth paths can be embedded. Neither path overlaps another so they are both added to the *left* stack (which corresponds, at the moment, to the inside face) and each to a new block.
- (c) shows the embedding of the fifth path; this path overlaps the fourth (red) path (but not the third) and results in the paths of that block being swapped from *left* to *right*, and vice versa, (the result of swapping the fourth path is shown in black) and then the fifth path being embedded to the *left* stack and to that swapped block.
- Considering (d): the algorithm backtracks to vertex *M*, the end of the fourth path which then has no further effect on the embedding process and is removed from the stack; at vertex *L*, the algorithm has backtracked to the start of the second path and all the blocks of branching paths (the third and fifth paths) are merged into the same block as the second path; upon backtracking to vertex *K*, the end of the fifth path is reached and it is removed from the stack; and then the sixth path is considered. The sixth path overlaps the third path (red) so the last block is swapped (moving the remaining second and third paths to the *right* stack and leaving the *left* stack empty) allowing it to be embedded.
- The cycle formed by the sixth path defines two faces (tan and brown shown in (e)) and within that the seventh paths can be embedded. Neither path overlaps another so they

are both added to the *left* stack (which corresponds, at the moment, to the inside face) and each to a new block.

- (f) shows the algorithm having backtracked past the start of the sixth path (so requiring the the block containing the seventh path to be merged into the previous block) and past the end of the third path (removing it from the stacks) then the eighth path can be added to the *left* stack, without overlap, and a new block.
- The final (ninth) path (g) overlaps the eighth path (requiring it to be swapped from the *left* to *right* stack and also overlaps the second path (which is already on the *right* stack, so no swap is required). After this, the ninth path can be embedded to the *left* stack and to the block formed by merging the previous two, since the both contained overlapping paths.
- Upon backtracking to vertex *A*, the algorithm completes as all paths will then have been removed from the stacks.

2.8.2 Extensions to H&T's algorithm

H&T's algorithm only tests whether a graph is planar or not. Extensions and revisions to this algorithm including:

- In 1977, Reingold, Nievergelt, and Deo [64] presented an updated version of the algorithm and a more detailed explanation of the testing process;
- In 1980, Williamson [83] presented an implementation based on H&T's algorithm and the work of Demoucron, Malgrange, and Petruiset [20] (who presents a non-linear time algorithm similar to Auslander and Parter, upon which H&T's algorithm is based, that explicitly considers the faces generated as successive super-graphs are formed by appending paths – there is an implicit relationship between the planarity testing process and the generated faces is H&T's algorithm but it is not explored);
- Williamson [84, 85] presented an extension to H&T's algorithm to extract Kuratowski sub-graphs upon failure of the planarity test;
- Kocay [41] gives an overview of H&T's algorithm and theoretical analysis of the properties of the branches of the DFS tree and proof (mostly independent of an algorithmic implementation) that the method tests correctly for planarity; and
- Mehlhorn, Mutzel, and Näher present a series of papers [52–54, 57] on their implementation of H&T's algorithm used in the LEDA graph library and its extension to generate a cyclic edge order for each vertex representing a planar embedding of the graph or, on failure, to extract a Kuratowski sub-graph.

2.8.3 “Complexity” of H&T's algorithm

Chiba et al. [14] commented that “*the modification [of H&T's planarity testing algorithm to construct an embedding] looks to be fairly complicated; in particular it is quite difficult to implement a part of the algorithm for embedding an intractable path called a special path*”. One of the main reasons for this issue is that, while the Left-Right stacks are so named, there is no correlation between a path's positioning on the Left (Right) stack and its position within an embedding.

This is highlighted in figures 2.8.6-2.8.8, below.

- Figure 2.8.7a shows the embedding of paths 1, 2 & 3 and figure 2.8.8a shows their corresponding position on the (Left) stack. The paths within the yellow and brown areas in the embedding correspond to the highlighted paths on the stacks.
- Figures 2.8.7b & 2.8.8b shows the embedding of paths 4 & 5 onto the Left stack; both of which conflict with path 3 causing it to be moved to the Right stack. The yellow and brown areas highlighted in the embedding are unchanged from the previous embeddings of paths 1-3 and still contain the correspond to the contents of the stacks.
- Figures 2.8.7c & 2.8.8c shows the embedding of path 6, which conflicts with path 5 and so paths 4 & 5 are both contained in the same block and are moved to the Right stack (the algorithm has backtracked to the end of path 3 and it is removed from the stack). In this case, since path 4 can only be within the face bounded by vertices A, B, C, D, E, F & G and is now contained in the Right stack, the stacks correspond to the opposite areas highlighted in the prior embedding stage.

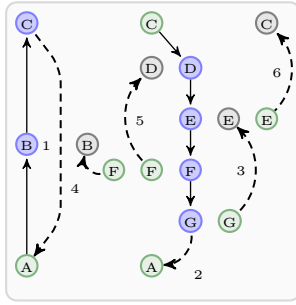


Figure 2.8.6: Second Example of Disjoint Paths (Numbered in Embedding Order) Generated by Second Phase of H&T's Planarity Test

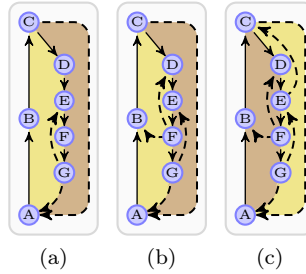


Figure 2.8.7: Second Example – Embedding of Successive Paths (Contents of Coloured Regions Correspond to Contents of Stacks in Figure 2.8.8)

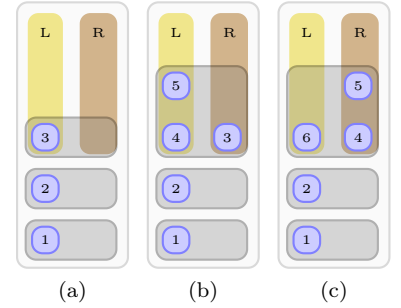


Figure 2.8.8: Second Example – Contents of Left-Right Stacks Corresponding to Successive Embeddings Showing Lack of Correlation Between Path's Stack and Embedding Region

Mehlhorn, Mutzel, and Näher [52–54, 57] show that this problem is not insurmountable. However, this issue has caused authors of alternate planarity tests to criticise the algorithm (whilst discussing their own works); this includes: Shih and Hsu [68, pg. 1], who states that H&T's algorithm is “quite involved” and that the “partial planar sub-graph construction [...] needs to be continually modified to obtain a final embedding”; and Boyer et al. [12, pg. 243], who note that the algorithm is “complex” (in reference to Chiba et al.'s quote).

2.9 Booth and Lueker's Vertex Addition Planarity Test – 1976

Booth and Lueker [5] published the second linear-time planarity test two years after Hopcroft and Tarjan. This improves upon Lempel, Even, and Cederbaum's [47] $O(V^2)$ algorithm through the introduction of an efficient *PQ Tree* data structure and the use of templates that restrict the permissible permutations and allows the graph to be tested for planarity such that “[i]ts time complexity is shown to be linear in the size of the input”.

Lempel, Even, and Cederbaum's algorithm uses Bush Form representations of successive *st*-ordered sub-graphs to test for planarity. These Bush Forms are separated into maximal biconnected or bridge edge pieces such that pairs of connected pieces are separated by an articulation vertex; the *PQ Tree* data structure corresponds to the Bush Form such that:

- The leaves of the *PQ Tree* represent the virtual vertices of the Bush Form;
- *P-nodes* represent articulation vertices, with two or more outbound pieces (children), or maximal biconnected pieces with exactly two outbound pieces (children); and
- *Q-nodes* represent maximal biconnected pieces, with three¹⁴ or more outbound pieces (children).

Corresponding to the transformations that can be performed on the Bush Form to reorder the virtual vertices, the following equivalence transformations are permitted on a *PQ Tree*:

- Permuting the order of the children of a *P-node*; or
- Reversing (flipping) the order of the children of a *Q-node*.

Booth and Lueker given an additional status, for a *PQ Tree* $\mathcal{T}_k$ equivalent to the Bush Form $\mathcal{B}_k$, such that: a leaf is *pertinent* if it is labelled $(k + 1)$; and a node is *full/partial/empty* if all/some/none (respectively) of its descendant leaves are *pertinent*. The order of children of nodes that are full or empty are inconsequential to determining whether the leaves labelled $(k + 1)$ have a consecutive ordering.

Using this, Booth and Lueker show that the test for planarity at any inductive step (of coalescing the next *st*-ordered leaves to add a vertex to the embedding) needs only consider the minimal sub-*PQ Tree* containing all *pertinent* leaves. To simplify the process, nine templates (patterns and corresponding replacements) are defined such that each node within that minimal sub-tree must match one of those patterns (otherwise the graph is non-planar) and the sub-tree can be reduced (and simplified) by replacing that pattern with its corresponding replacement. To create the *PQ Tree* $\mathcal{T}_{k+1}$ from $\mathcal{T}_k$ then $\mathcal{T}_k$ is first reduced using the templates and then all the (now consecutive) leaves labelled $(k + 1)$ are replaced by a *P-node* whose children are leaves corresponding to its adjacent vertices labelled $(> k + 1)$.

Chiba et al. [14] augment Booth and Lueker's algorithm to generate a planar embedding of the graph, also in linear time.

¹⁴ [5, pg. 339] “[T]here is no real distinction between a *P-node* and a *Q-node* if there are only two children. It is convenient to remove this redundancy”; “[...] guaranteeing unique *PQ-tree* representations for the classes of permutations being studied.”

2.10 Shih-Hsu – Edge Addition Planarity Test 1999

Shih and Hsu [69] introduce a variation on the *PQ Tree* data structure called a *PC Tree* and a test for planarity by successively appending edges to the embedding.

The algorithm pre-processes the graph using Depth-First Search: partitioning the edges into *tree* and *cotree* (back) edges; assigning each vertex an ascending index, in post-DFS order; and applies an ordering the edges.

The main body of the algorithm begins by embedding tree edges then backtracking up the DFS-tree until a vertex with an inbound cotree edge is reached. The algorithm then tests against a number of patterns, each defining non-planarity characteristics, and if none are matched then applies a reduction to embed the inbound cotree edge(s); this forms a biconnected component where:

- The vertices adjacent only to interior faces of that component have all their incident edges processed and their embedding has been found to be planar so can be ignored;
- The vertices on the boundary of the external face of that component form a cycle which can be represented by a *C-node* (defining the cyclic order of those vertices);
- Vertices in the *C-node* with incident unprocessed edges are represented by *P-nodes*, attached to the *C-node*, and allowing the order of those unprocessed edges to be permuted;

As branches in the DFS-tree are explored disjoint biconnected components can be formed and, when the algorithm backtracks beyond a branching point of the DFS-tree which joins two or more biconnected components, and inbound cotree edges are found, then these components can be coalesced. Similarly to Booth and Lueker's algorithm templates are used to restrict the order of *P-nodes* and biconnected components which do not match a pattern are non-planar.

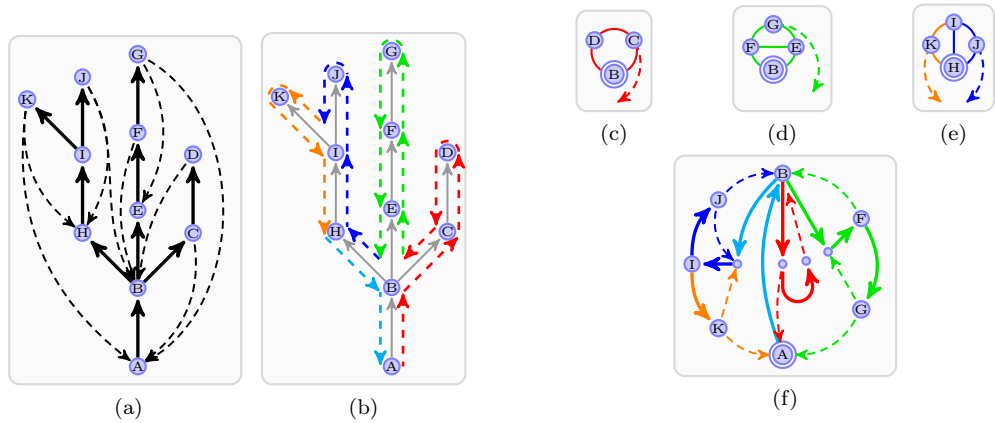


Figure 2.10.1: Example of the Generation of C-nodes and Their Concatenation

Figure 2.10.1 shows: an example graph (a); the path followed by the DFS algorithm (b); the biconnected components (c)/(d)/(e) formed after the DFS processes the red/green/dark blue & orange paths (respectively) of (b) (the root of the DFS-tree for these components is shown as a node with a double circle); and (an embedding of) the biconnected component (f) formed from processing the light blue path and coalescing the previously generated components.

The PC-Tree algorithm was originally formulated by Shih and Hsu [68] in 1993, however Boyer et al. note [12, pg. 130] that this original “formulation lacks a description of how to implement the routines that manipulate the PC-tree to solve the planarity problem”. Thomas [75] presented a self-contained exposition on this methodology producing an (correctly formulated) algorithm that (in his words) “*is not completely satisfactory*” as it requires the (sub-) graph being tested to be 3-connected. Later papers by Shih and Hsu [33–36, 69] suggest clarifications and improvements to the algorithm but, as noted by Haeupler and Tarjan [29, pg. 2], “*gave incorrect algorithms*”. A corrected algorithm was finally published by Boyer et al. [11] (building on the earlier works of both Shih and Hsu and Thomas) and additional details of the non-planarity tests given in Boyer.

2.11 Boyer-Myrvold Planarity Test

In 1999, Boyer and Myrvold [8] published a modification of Booth and Lueker’s PQ-Tree algorithm which uses a similar methodology to Shih and Hsu’s algorithm [68]; a high-level overview of the algorithm is given in 2.11.1, below (from [9, pg. 8]). This has been improved over a series of papers [9, 10, 12, 13]

Listing 2.11.1: High-Level Outline of Boyer-Myrvold Planarity Test

- 1 (1) Receive graph $\mathcal{G}$ with $n > 2$ vertices and $m \leq 3n - 5$ edges
- 2 (2) Perform depth first search on $\mathcal{G}$ and other preprocessing
- 3 (3) Based on $\mathcal{G}$, create and initialise embedding $\bar{\mathcal{G}}$
- 4 (4) Add each DFS tree edge of $\mathcal{G}$ to $\bar{\mathcal{G}}$ as a singleton biconnected component
- 5 (5) For each vertex v of $\mathcal{G}$ in reverse DFI order
- 6 (6) Embed in $\bar{\mathcal{G}}$ each back edge in $\mathcal{G}$ from v to a DFS descendant of v .
 For each such back edge (v, w) , embed (v, w) such that:
 - 7 a) all biconnected components are merged together that will no longer
 be separable then (v, w) is added;
 - 8 b) any vertex x with unembedded back edges to v or DFS ancestors of v
 is kept on the external face (along with cut vertices separating x from
 9 v).
 - 10 If embedding (v, w) requires violation of 6b, break the loop.
- 11 (7) If one or more back edges were not embedded, Isolate a Kuratowski subgraph.

Boyer and Myrvold use a similar (to Shih and Hsu’s algorithm) reverse-DFS order to process vertices and to embed successive edges; however, they use a PQ-tree structure modified such that a Q-node is represented by a doubly-linked cycle (equivalent to a C-node in a PC-tree) which is rooted at the vertex with minimum DFS index, restricting the cycle from rotating but allowing it to be flipped – simplifying the testing process. The algorithm involves iteratively: testing embedded biconnected components for non-planarity patterns; embedding edges onto biconnected components; and if necessary, (flipping and) merging pairs of biconnected components.

Boyer and Myrvold assert, based on empirical testing in [10], that (alongside [17]’s algorithm [17]) their implementation of this algorithm is one of the fastest implementations of a planarity testing algorithm.

2.12 De Fraysseix and Rosenstiehl's Trémaux Tree Planarity Characterisation and Test – 1982-2008

In 1982, De Fraysseix and Rosenstiehl [18] gave a characterisation of planarity in terms of Trémaux (Depth-First Search) Trees that presented five patterns for the arrangement of cotree edges incident to the spanning tree of a Trémaux Tree. This was based on an earlier paper by Rosenstiehl [67] who in turn drew upon an earlier planarity criterion of Wu [87] and a planarity test of Liu [49] [50, partially summarised in]; they later [19] simplified this characterisation reducing it from five to three templates. De Fraysseix and Rosenstiehl note that Hopcroft and Tarjan's proof of their planarity test depends on “*a lemma associated with the computing process*” instead of a formal definition on the structure of the paths and this characterisation fills this gap.

The characterisation considers the equivalence of angles formed cotree edges with the Trémaux Tree; it defines, based on precedence of vertices within the Trémaux Tree, three configurations (Figure 2.12.1) of edges (marked α and β) such that that a pair of angles (highlighted in red and blue) are either classed as being the “*T-alike*” (a) or “*T-opposite*” (b and c) depending on whether they are on the same or opposing sides of the tree. A graph is non-planar if there exists a pair of edges which are characterised as both “*T-opposite*” and “*T-alike*”.

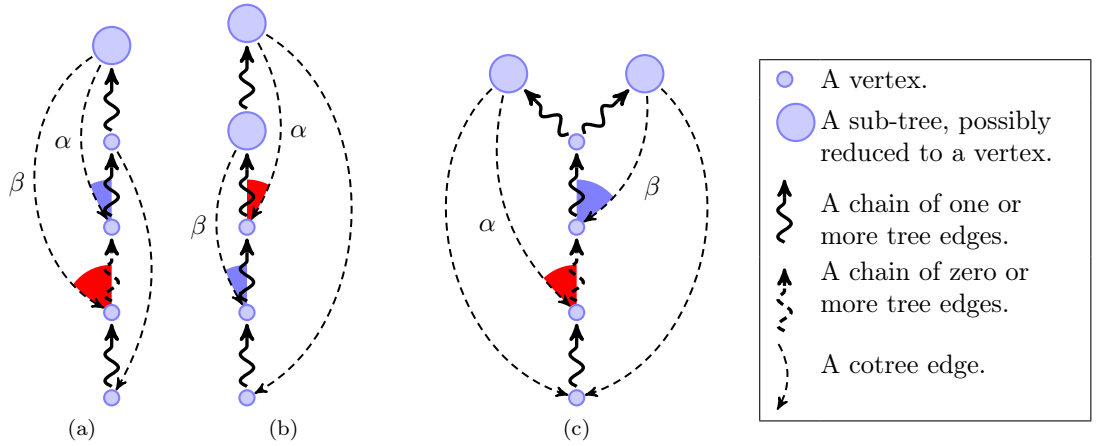


Figure 2.12.1: De Fraysseix and Rosenstiehl's Planarity Characterisation on Trémaux Trees

In 2006, de Fraysseix, Ossona de Mendez, and Rosenstiehl [17] published a planarity testing algorithm; then in 2008 de Fraysseix [15] published a simplification. Comparing this algorithm to H&T's:

- Hopcroft and Tarjan pre-processes the graph to generate vertex and edge orderings before separating the graph into a cycle and a series of paths such that each of these paths is terminated by a back (cotree) edge. They then consider the embedding of these paths in a specific order and use two stacks to enforce the condition where paths must be embedded on the same or opposite sides; testing for these conditions is performed by comparing the precedence within the DFS (Trémaux) tree of the vertex those paths (and, correspondingly, back edges) are inbound to.
- De Fraysseix uses a pre-processing phase (identical to H&T) to generate: an order over the vertices; directions for the edges; and calculations of low points; before creating an (again, identical to H&T) order on the outbound edges. He then, similarly, considers the embedding of the cotree edges (using this identical order) to generate an f -colouring which

enforces the condition that cotree edges must be *T-alike* or *T-opposite* at the vertex those cotree edges are inbound to.

In both the 2006 and 2008 papers, the first phase of the algorithm is presented in great detail using mathematical notation whereas the second phase is sketched in less than a page. An implementation of the algorithm is included in the PIGALE library [16], however (as of version 1.3.9) the documentation appears to consist solely of class dependency diagrams and the source code is uncommented. The lack of detail and clarity in all these aspects makes it very difficult to know exactly how the algorithm functions and whether it is an optimised (or simplified) version of H&T's algorithm or if it contains a novel approach.

In either case, this does not detract from the detailed mathematical characterisation of Trémaux Trees that the authors have presented and the fact that, during testing of the PIGALE and LEDA libraries, Boyer et al. [12] noted that this algorithm provided the fastest test for planarity of all implementations within those libraries.

2.13 Planarity Algorithms via PQ-Trees – 2008

Bernhard Haeupler [29] presents¹⁵ a new algorithm (developed in conjunction with Robert Tarjan) based on a simplification of the PQ and PC methodology showing that the two data structures are different implementations of the same methodology.

The algorithm presented by Haeupler creates one or more PQ-Tree style structure where vertices can be fully, partially or not embedded. Fully embedded vertices are within the PQ-Tree, partially embedded vertices are at the leaves of the tree and the non-embedded vertices are awaiting addition to the tree. As a vertex is added to the structure if it connects to two or more PQ-Trees the structures are combined but permutations are not discarded unless one permutation would invalidate the planarity of the graph.

The extended abstract and presentation give an overview of the methodology but does not give a concrete definition of the algorithm and while the process looks promising there is not enough detail to fully evaluate the algorithm.

One of the interesting side points in Haeupler's conference presentation was the side note that there is no current path addition algorithm that can give all possible embeddings of a planar graph.

2.14 Conclusions

There are relatively few papers in the field of planarity testing and embedding as optimal algorithms rely on a few basic concepts (DFS-trees or BFS-trees) and it is rare that a significantly different approach to the field is found so most papers contain simplification or optimisations of a previous approach.

In the field of Path addition the first algorithm by Hopcroft and Tarjan [32] and the later embedding process by Mehlhorn and Mutzel [52] are still state-of-the-art in this field. The work of de Fraysseix [15] is similar (and may be a simplification of Hopcroft and Tarjan's algorithm or may be a novel algorithm that uses some similar concepts) and has proved [12] itself to be a fast algorithm.

Vertex and Edge addition are the major areas of research as the data structures are improved or refined and the algorithm by Boyer and Myrvold [11] is considered to be the state-of-the-art in this category of planarity testing algorithms. The latest work by Haeupler and Tarjan [29] may supersede this and give a vertex addition method that generalises PQ and PC trees and allows all possible embeddings of a planar graph to be generated but as the extended abstract only presents an overview of the processes used there is not an algorithm to evaluate (nor has it appeared to be forthcoming).

Given this overview of the field it is clear that there is an opening for an algorithm that can generate an embedding of a graph, where possible, and to give all possible embeddings of a planar graph in linear time. The final part of this is new and novel and does not appear in any of the previous algorithms.

¹⁵Video of the presentation is available from <http://www.canalc2.tv/video.asp?idvideo=7543>

Chapter 3

Trémaux Trees

Trémaux Trees are so named after Trémaux’s (Depth-First Search) algorithm which generates a rooted directed spanning tree with specific properties (described later in this chapter) as it processes an (undirected connected) graph. Various papers presented by W. Wu [87], Pierre Rosenstiehl [67], Hubert de Fraysseix et al. [15, 17–19], and Yanpei Liu [50] present definitions of Trémaux Trees and the statement that a DFS algorithm generates a Trémaux Tree. Proof of this statement (although not couched in terms of Trémaux Trees) can be found in, amongst others, undergraduate textbooks such as [4, Pg. 141-142].

3.1 Trémaux Tree Definitions

The definition of a Trémaux Tree and subsequent definitions (3.1.1 – 3.1.11) relating to the structure of the Trémaux Tree are included below (for completeness). The final pair of definitions in this section (3.1.12 & 3.1.13) are included as a clarification to de Fraysseix’s work [15, 17] regarding conversion of a partial order of the set of edges outbound from any given vertex to a total order and have particular reference in later chapters when considering permutations of embeddings.

DEFINITION 3.1.1. A Trémaux Tree (TT), $\mathcal{T}(\mathcal{G}, \mathcal{F}, r)$, defines a partition on the edge-set ($\mathcal{E}$) of a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ into the set of tree edges ($\mathcal{F}$) and cotree edges ($\mathcal{E} \setminus \mathcal{F}$) such that:

- The tree edges form a rooted spanning tree of $\mathcal{G}$ (rooted at vertex r);
- A partial order relationship “precedes or equals” ($\preceq$) can be defined on $\mathcal{V}$ such that $x \preceq y$ if the path within the rooted spanning tree, defined by $\mathcal{F}$, from r to y includes x (and $x \prec y \Leftrightarrow x \preceq y \wedge x \neq y$);
- $\mathcal{T}$ is a Trémaux Tree if, and only if, every edge of $\mathcal{G}$ is incident to two comparable ($\perp$) vertices ($u \perp v \Leftrightarrow u \preceq v \vee v \preceq u$).

DEFINITION 3.1.2. Each edge $\{u, v\} \in \mathcal{E}$ of a Trémaux Tree can be given directionality $(u \rightarrow v)$ where, without loss of generality:

- If $\{u, v\} \in \mathcal{F}$ then $u \prec v$; and
- If $\{u, v\} \in \mathcal{E} \setminus \mathcal{F}$ then $v \preccurlyeq u$.
- $(u \xrightarrow{\mathcal{T}} v)$ denotes a directed tree edge.
- $(u \xrightarrow{\mathcal{C}} v)$ denotes a directed cotree edge.

COROLLARY 3.1.3. There is a bijection from the set of tree edges and the set of non-root vertices $(\forall v \in \mathcal{V} \setminus \{r\}, \exists! u \in \mathcal{V} : (u \xrightarrow{T} v) \in \mathcal{F})$.

DEFINITION 3.1.4. The partial order $\preccurlyeq$ can be extended to $\mathcal{V} \cup \mathcal{E}$ such that:

- The “immediately precedes” ($\prec_{\text{im}}$) relationship can be defined such that $u \prec_{\text{im}} v \Leftrightarrow (u \prec v \wedge (\nexists x \in \mathcal{V} \cup \mathcal{E} : u \prec x \prec v))$.
- For all $(u \xrightarrow{T} v)$: $u \prec_{\text{im}} (u \xrightarrow{T} v) \prec_{\text{im}} v$.
- For all $(u \xrightarrow{C} v)$: $v \preceq u \prec_{\text{im}} (u \xrightarrow{C} v)$ which terminates the branch of the

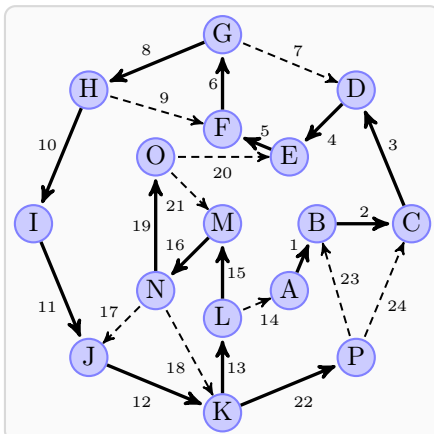


Figure 3.1.1: Example Graph with Edges Labelled by Depth-First Search Pre-Order (Tree edges are thick, cotree edges are dashed).

Trémaux Tree (such that $\nexists x \in \mathcal{V} \cup \mathcal{E} :$
 $(u \xrightarrow{C} v) \prec x$).

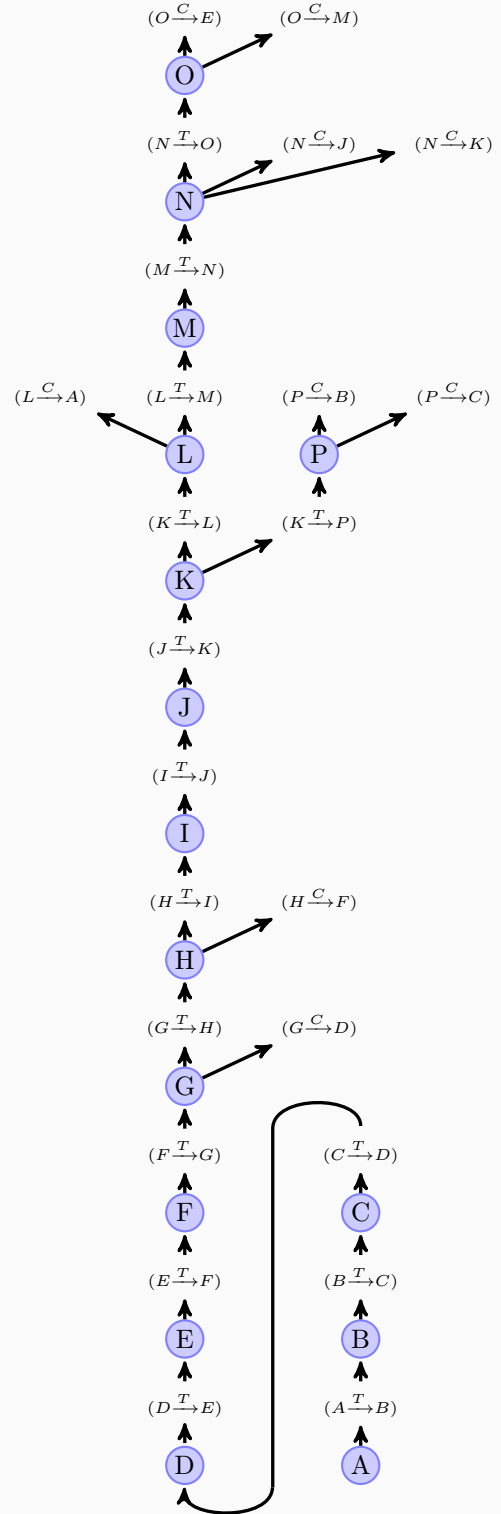


Figure 3.1.2: Trémaux Tree of Example Graph (Figure 3.1.1) where each connection in the tree represents a $\prec$ relationship

DEFINITION 3.1.5. Each edge's direction defines the vertices it is inbound to and outbound from; for each edge $e \in \mathcal{E}$ where $e = (u \rightarrow v)$ then $\text{in}(e) = v$ and $\text{out}(e) = u$.

DEFINITION 3.1.6. The edges' directions define mappings from each vertex to the sets of edges outbound from and inbound to that vertex; for each vertex $v \in \mathcal{V}$:

- $\text{outbound}(v) = \{e \in \mathcal{E} : \text{out}(e) = v\}$
- $\text{inbound}(v) = \{e \in \mathcal{E} : \text{in}(e) = v\}$

DEFINITION 3.1.7. The reachable elements (vertices and edges) of $\mathcal{G}$ from a given element x is the union of the set of elements succeeding or equalling x (the sub-tree of the Trémaux Tree rooted at x) and the set of vertices connected to edges within that sub-tree (which is: all vertices in the sub-tree; the vertices cotree edges in that sub-tree are inbound to; and if x is an edge then the vertex which x is outbound from):

- $\text{reachable}(x) = \{y \in \mathcal{V} \cup \mathcal{E} : x \preceq y\} \cup \{\text{in}(e), \text{out}(e) \in \mathcal{V} : e \in \mathcal{E}, x \preceq e\}$
 $= \{y \in \mathcal{V} \cup \mathcal{E} : x \preceq y\} \cup \{x'\} \cup \{\text{in}(z) : z \in \mathcal{E} \setminus \mathcal{F}, \text{in}(z) \preceq x \preceq z\}$
 where $x' = \begin{cases} x & : x \in \mathcal{V} \\ \text{out}(x) & : x \in \mathcal{E} \end{cases}$

DEFINITION 3.1.8. The low-points of a given element x of a Trémaux Tree are the reachable vertices which equal or precede x :

- $\text{lowPoints}(x) = \{y \in \text{reachable}(x) \cap \mathcal{V} : y \preceq x\}$
 $= \{x'\} \cup \{\text{in}(z) : z \in \mathcal{E} \setminus \mathcal{F}, \text{in}(z) \preceq x \preceq z\}$ where $x' = \begin{cases} x & : x \in \mathcal{V} \\ \text{out}(x) & : x \in \mathcal{E} \end{cases}$
- $\text{lowPoints}(x)$ is a subset of the unique path, within $\mathcal{F}$, from the root vertex to x .

Additionally, each element x has at least one low-point vertex x' and $\max(\text{lowPoints}(x)) = x'$.

DEFINITION 3.1.9. The first low-point vertex is defined as the low-point vertex with minimum precedence.

- $\text{lowPoint1}(x) = \min(\text{reachable}(x)) = \min(\text{lowPoints}(x))$

DEFINITION 3.1.10. The second low-point vertex is defined as the low-point vertex with the second lowest precedence; if an element only has a single low-point vertex then the second low-point vertex is defined to be that (maximum precedence) vertex.

- $\text{lowPoint2}(x) = \min((\text{reachable}(x) \setminus \text{lowPoint1}(x)) \cup \{\max(\text{lowPoints}(x))\})$
 $= \min((\text{lowPoints}(x) \setminus \text{lowPoint1}(x)) \cup \{x'\})$ where $x' = \begin{cases} x & : x \in \mathcal{V} \\ \text{out}(x) & : x \in \mathcal{E} \end{cases}$

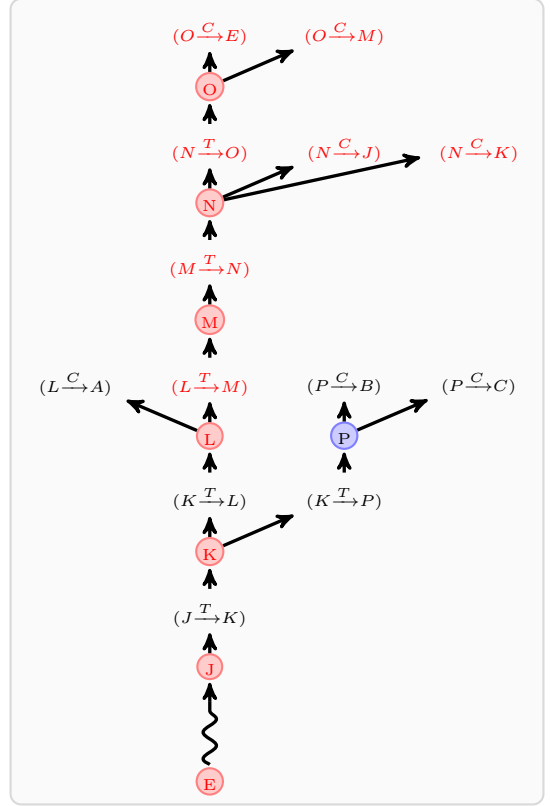


Figure 3.1.3: Reachable vertices and edges (red) from edge $(L \xrightarrow{T} M)$ for the Trémaux Tree (Figure 3.1.2) of Example Graph

Figure 3.1.3 shows the elements of the example graph (Figure 3.1.1) that are reachable from edge $(L \xrightarrow{T} M)$; the first and second low-points of edge $(L \xrightarrow{T} M)$ are vertices E and J , respectively.

DEFINITION 3.1.11. A TT-precedence order $\prec^*$ is a partial order defined on the disjoint sets of edges outbound from each vertex. For any $v \in \mathcal{V}$ and $e_1, e_2 \in \text{outbound}(v)$ then $e_1 \prec^* e_2$ if:

- $\text{lowPoint1}(e_1) \prec \text{lowPoint1}(e_2)$; or
- $\text{lowPoint1}(e_1) = \text{lowPoint1}(e_2) \wedge \text{lowPoint2}(e_1) = v \wedge \text{lowPoint2}(e_2) \prec v$

DEFINITION 3.1.12. For any $v \in \mathcal{V}$ and $e_1, e_2 \in \text{outbound}(v)$ where $\prec^*$ does not define an ordering then those edges are defined to have an incomparable TT-precedence order ($\parallel^*$). This occurs when $\text{lowPoint1}(e_1) = \text{lowPoint1}(e_2)$ and either:

- $\text{lowPoint2}(e_1) = \text{lowPoint2}(e_2) = v$; or
- $\text{lowPoint2}(e_1) \prec v \wedge \text{lowPoint2}(e_2) \prec v$

Figure 3.1.4 (b) shows examples of both cases of incomparable TT-precedence edges: the red edges are examples of the first case (where the edges' second low-points are equal to the vertex the edges are outbound from); and the green edges are examples of the second case (where the edges' second low-points precede the vertex the edges are outbound from).

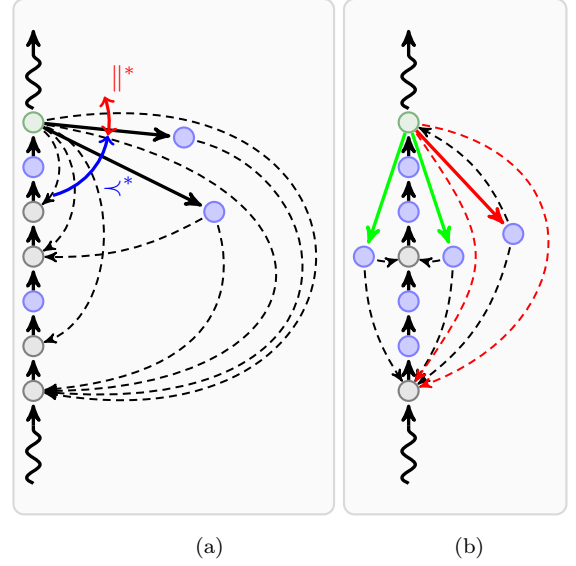


Figure 3.1.4: (a) Edges outbound from the same vertex in TT-precedence order from the three outermost edges with (incomparable) lowest precedence to the innermost with highest precedence. (b) Two sets (red and green) of incomparable precedence edges outbound from the same vertex – the green edges have higher precedence than the red edges.

DEFINITION 3.1.13. For each vertex's set of outbound edges, the partial order $\prec^*$ can be extended to a total order ($<^{**}$) such that each sub-set of edges with incomparable TT-precedence ($\parallel^*$) is assigned an arbitrary ordering.

Let $\{e_1, e_2, e_3, \dots, e_n\} = \text{outbound}(v)$ such that $e_1 <^{**} e_2 <^{**} e_3 <^{**} \dots <^{**} e_n$ then:

- $\forall e_i, e_j \in \text{outbound}(v) : i < j \Rightarrow (e_i \prec^* e_j \vee (e_i \parallel^* e_j \wedge e_i \text{ is arbitrarily ordered before } e_j))$
- $\min^{**}(\text{outbound}(v)) = e_1$ and $\max^{**}(\text{outbound}(v)) = e_n$

This final pair of definitions (3.1.12 & 3.1.13) are included as a clarification to the definitions in de Fraysseix's paper [15]. De Fraysseix notes that the $\prec^*$ relationship (def. 3.1.11) is a partial order [15, pg. 172] but uses it to define a total order [15, pg. 175] on the set of outbound edges of each vertex.

This is a minor point, with regards to de Fraysseix's paper, as a software implementation of de Fraysseix's algorithm will contain an underlying data structure used to store edges outbound from a given vertex (maintaining a static order for those edges) and the combination of the $\prec^*$ partial order and the underlying data structure's order implicitly creates a total order ($<^{**}$). However, this point is central to identifying alternate embeddings of a graph since varying (and controlling) the order of the edges (related via $\parallel^*$) in the underlying data structure will vary (and control) the order in which those edges appear in the total order – definition 3.1.13 explicitly captures this relationship.

Chapter 4

Planarity Testing

4.1 Segments of a Trémaux Tree

The H&T planarity testing algorithm [32] considers, for a biconnected graph, a cycle containing the root vertex; removal of this cycle separates the Trémaux (DFS) Tree into the cycle and a forest of disjoint sub-trees emanating from the cycle. Each sub-tree contains a path from the minimum precedence edge of that sub-tree to that edge's first low point; removal of this path separates the sub-tree into further disjoint sub-trees. Recursively considering sub-trees then a path can be removed from each sub-tree separating it into successively smaller until each sub-tree is reduced to nothing and the graph can be considered to be composed of the set of (removed) disjoint paths; upon which planarity can be tested.

H&T define a segment of the graph to be one of the recursively generated sub-trees (of the Trémaux Tree) and refer to the eventual output simply as paths. This use of terminology is solely related to the theory supporting the algorithm's correctness as to achieve a linear execution time the disjoint paths are generated directly and no data structure is generated to represent the segments; as such the paths generated by H&T's algorithm have a hierarchical nature corresponding to the containing nested sub-trees but this is not explicitly noted by H&T. To try to capture these properties, the "segment" terminology is alternately defined to be a path generated by this process (and so has specific properties, as described below) and the more generic terms of path and sub-tree can be used where its traditional meaning is expected. In addition, a generic term "chain" is also defined specifically relating to a path within a Trémaux Tree as this is useful in defining a segment and in later analysis of the properties of segments.

The closed interval notation can be used [15, pg. 171] to represent a chain of elements (a simple path of elements all related within the Trémaux Tree):

DEFINITION 4.1.1. A chain of elements with minimum and maximum precedence elements u and v , respectively, (denoted by $[u; v]$) is defined as the (maximal) ordered (via $\prec$) set containing all elements x in the Trémaux Tree where $u \preceq x \preceq v$.

- Let $\langle \mathcal{A}, \mathcal{R} \rangle$ denote the sequence (totally ordered set) containing all elements in set $\mathcal{A}$ ordered by the relationship $\mathcal{R}$ then a chain can be equivalently denoted by:

$$[u; v] \quad \equiv \quad \langle \{ \forall x \in \mathcal{V} \cup \mathcal{E} : u \preceq x \preceq v \}, \prec \rangle.$$

- Additionally, the following notations can be defined:

$$]u; v] \quad \equiv \quad \langle \{ \forall x \in \mathcal{V} \cup \mathcal{E} : u \prec x \preceq v \}, \prec \rangle;$$

$$\begin{aligned} [u; v[&\equiv \langle \{\forall x \in \mathcal{V} \cup \mathcal{E} : u \preceq x \prec v\}, \prec \rangle; \text{ and} \\]u; v[&\equiv \langle \{\forall x \in \mathcal{V} \cup \mathcal{E} : u \prec x \preceq v\}, \prec \rangle. \end{aligned}$$

- The head (tail) of the chain is defined as the element within the chain with minimum (maximum) TT-precedence:

$$\begin{aligned} \text{head}([u; v]) &= u; \text{ and} \\ \text{tail}([u; v]) &= v. \end{aligned}$$

A definition of a segment (defining equivalent paths to those generated by pathfinder algorithm used by H&T [32, pg. 556]) can be made using the concept of a chain within a Trémaux Tree:

DEFINITION 4.1.2. The vertices and edges of $\mathcal{G}$ can be partitioned into a set of segments such that:

- Each segment is a chain;
- The segments are disjoint and cover all elements of the graph – each vertex and edge is contained in exactly one segment;
- For all $(u \xrightarrow{\mathcal{T}} v) \in \mathcal{F}$ then $(u \xrightarrow{\mathcal{T}} v)$ and v are contained in the same segment;
- For all $v \in \mathcal{V}$ where $\text{outbound}(v) \neq \emptyset$ then v and the minimal TT-precedence edge outbound from v ($\min^{**}(\text{outbound}(v))$) are contained in the same segment; and
- Each non-minimal TT-precedence outbound edge is at the head of a distinct segment.

The concept of head and tail of each segment can be extended such that:

- The head (tail) edge of a segment is the edge with the minimum (maximum) precedence within that segment:

$$\begin{aligned} \text{headEdge}(s) &= \min(s \cap \mathcal{E}); \text{ and} \\ \text{tailEdge}(s) &= \max(s \cap \mathcal{E}). \end{aligned}$$

- The head (tail) vertex of a segment is defined as the vertex the head (tail) edge is outbound from (inbound to):

$$\begin{aligned} \text{headVertex}(s) &= \text{out}(\text{headEdge}(s)); \text{ and} \\ \text{tailVertex}(s) &= \text{in}(\text{tailEdge}(s)). \end{aligned}$$

- For example, if the head edge is $(u \rightarrow v)$ and the tail edge is $(x \rightarrow y)$ then the head vertex is u and the tail vertex is y .

Given this definition then the following relationships can be defined between segments:

DEFINITION 4.1.3. The "is an ancestor of" relationship can be defined on the set of segments ($\mathcal{S}$) such that segment s_α is an ancestor of segment s_β if, and only if, $s_\alpha \neq s_\beta$ and s_α contains any element (vertex or edge) α and s_β contains any element β such that $\alpha \prec \beta$.

The set of ancestors of a given segment is defined to be the maximal set of segments which are each an ancestor of that segment.

COROLLARY 4.1.4. If s_α is an ancestor of s_β then there must exist elements $\alpha \in s_\alpha$ and $\beta \in s_\beta$ where $\alpha \prec \beta$. Given that segments are disjoint and maximal (with regards to set inclusion of $\prec$ related elements between its head and tail – definition 4.1.2) then $\text{head}(s_\alpha) \preceq \alpha \prec \text{head}(s_\beta) \preceq \beta$ therefore s_α is an ancestor of s_β if, and only if, $\text{head}(s_\alpha) \prec \text{head}(s_\beta)$.

COROLLARY 4.1.5. The "is an ancestor of" relationship is anti-symmetric and transitive – this follows directly from the fact that the $\preceq$ relationship is a partial order (reflexive, anti-symmetric and transitive) relationship and that if s_α is an ancestor of s_β then:

- $s_\alpha \neq s_\beta$ and so the "is an ancestor of" relationship is not reflexive; and
- The elements at the head of s_α and s_β are related via $\prec$ so, it still holds from this relationship that, the "is an ancestor of" relationship is anti-symmetric and transitive.

For example, if segment s_n has ancestors $\{s_1, s_2, s_3, \dots, s_{n-1}\}$ and is preceded by the chain $[r; \text{head}(s_n)[$ which can be split into disjoint sub-chains such that $[r; \text{head}(s_2)[\subset s_1$, $[\text{head}(s_2); \text{head}(s_3)[\subset s_2$, $[\text{head}(s_3); \text{head}(s_4)[\subset s_3 \dots$ and $[\text{head}(s_{n-1}); \text{head}(s_n)[\subset s_{n-1}$ then s_1 is an ancestor of s_2 which are ancestors of s_3 which are ancestors of $\dots$ which are ancestors of s_{n-1} which are all ancestors of s_n .

DEFINITION 4.1.6. The "is a descendant of" relationship is defined such that segment s_α is a descendant of segment s_β if, and only if, s_β is an ancestor of s_α .

The set of descendants of a given segment is defined to be the maximal set of segments which are each a descendant of that segment.

DEFINITION 4.1.7. The "is a parent of" relationship is defined such that the parent of segment s is the ancestor of segment s where all other ancestors of s are also ancestors of that parent segment.

- $\text{parent}(s) = s_p : s_p \in \text{ancestors}(s), \forall s_a \in \text{ancestors}(s) \setminus \{s_p\}, s_a \in \text{ancestors}(s_p)$

DEFINITION 4.1.8. The "is a child of" relationship is defined such that segment s_α is a child of segment s_β if, and only if, s_β is a parent of s_α .

The set of children of a given segment is defined to be the maximal set of segments which are each a child of that segment.

- $\text{children}(s) = \{s_c \in \text{descendants}(s) : \text{parent}(s_c) = s\}$

DEFINITION 4.1.9. If two segments have the same parent then they are defined to be siblings.

DEFINITION 4.1.10. The segment containing the root vertex of the Trémaux Tree is defined to be the root segment and has the following properties:

- No element of the Trémaux Tree precedes the root vertex so the root segment has no ancestors and, hence, has no parent so is not a descendant, child or sibling of any other segment.
- The root vertex precedes all other elements of the Trémaux Tree so all other segments are descendants of the root segment; hence, each non-root segment has at least one ancestor (the root segment) and has a parent segment.

Considering the composition of segments:

DEFINITION 4.1.11. The leaves of the Trémaux Tree relative to any given element x are defined as the set of elements succeeding or equal to x with no successors (i.e. all cotree edges and vertices with no outbound edges):

- $\text{leaves}(x) = \{y \in \mathcal{V} \cup (\mathcal{E} \setminus \mathcal{F}) : x \preceq y, (\nexists z \in \mathcal{V} \cup \mathcal{E} : y \prec z)\}$
- $\forall x \in \mathcal{V} \cup \mathcal{E} : \emptyset \subset \text{leaves}(x) \subseteq \text{leaves}(r)$
- The leaves of a Trémaux Tree are defined as the maximal set of leaves for that Trémaux Tree – i.e. the leaves of the tree's root vertex ($\text{leaves}(r)$).

LEMMA 4.1.12. There is a bijection (one-to-one correspondence) from the set of leaves of the Trémaux Tree to the set of segments such that each leaf is the tail element of exactly one segment and the tail of each segment is a leaf.

PROOF 4.1.13. Given a segment containing a leaf element then that leaf and all other elements in the segment are related via $\prec$ (def. 4.1.1) and since no element succeeds a leaf (def. 4.1.11) then every leaf element is the tail element of its containing segment and no segment can contain multiple leaves.

If the tree edge $(u \xrightarrow{T} v)$ is contained in a segment then so is vertex v (def. 4.1.2) and since $u \prec (u \xrightarrow{T} v) \prec v$ (def. 3.1.4) then no tree edge $(u \xrightarrow{T} v)$ can be the tail of a segment.

If a vertex u with outbound edges is contained in a segment then so is the edge $(u \rightarrow v)$ with minimal TT-precedence of outbound (u) (def. 4.1.2) and since $u \prec (u \rightarrow v)$ (def. 3.1.4) then no vertex with outbound edges can be the tail of the segment.

Since no segment can have a tree edge or vertex with outbound edges as its tail then its tail must be either a cotree edge or a vertex with no outbound edges – these elements have no successors within, and are leaves of, the Trémaux Tree (def. 4.1.11).

Since: each leaf of the Trémaux Tree is contained in exactly one segment (def. 4.1.2); each segment's tail is a leaf; and segments cannot contain multiple leaves; then there is a one-to-one correspondence between leaves and segments. $\square$

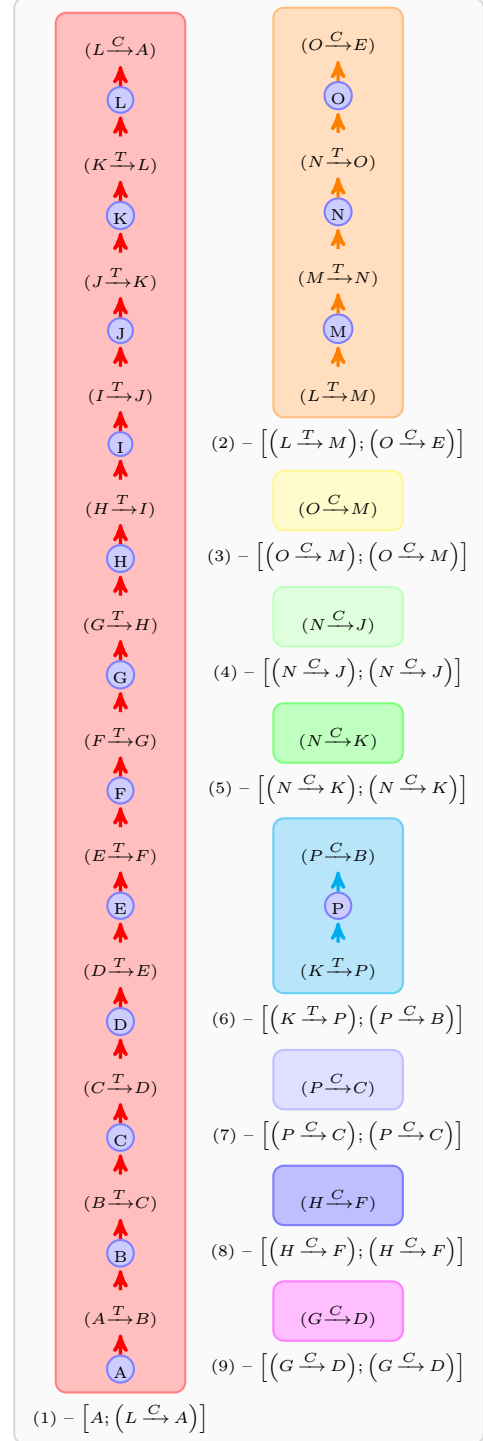


Figure 4.1.1: Nine segments (shown as coloured groups) which comprise the Trémaux Tree of Example Graph (Figure 3.1.1)

DEFINITION 4.1.14. The set of segments can be partitioned into four distinct classes such that, given a segment s with tail edge $(u \rightarrow v)$, the classes can be defined by the precedence of the tail vertex v :

Class 1: $\text{headVertex}(s) \prec (u \xrightarrow{T} v) \prec v$
(figure 4.1.2a);

Class 2: $\text{headVertex}(s) \prec v \prec (u \xrightarrow{C} v)$
(figure 4.1.2b);

Class 3: $v = \text{headVertex}(s) \prec (u \xrightarrow{C} v)$
(figure 4.1.2c); or

Class 4: $v \prec \text{headVertex}(s) \prec (u \xrightarrow{C} v)$
(figure 4.1.2d).

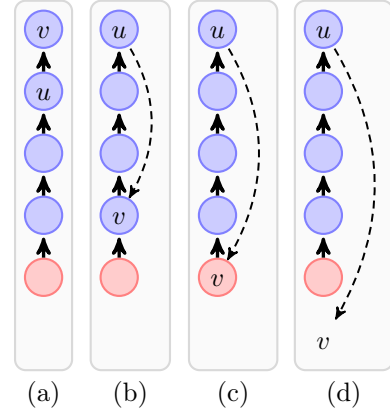


Figure 4.1.2: Examples of Segment Classes Based on Tail Vertex Precedence (Head Vertex in Red)

LEMMA 4.1.15. The head of each segment of a Trémaux Tree is:

- The root vertex – for the root segment;
- A tree edge – for non-root segments containing vertices; or
- A cotree edge – for non-root segments containing no vertices.

PROOF 4.1.16. The root vertex r precedes all other vertices and edges within the Trémaux Tree (def. 3.1.1 and 3.1.4) and since the head of a segment is the element with minimal precedence then the root vertex is always at the head of the segment containing it.

Each tree edge, $(u \xrightarrow{T} v) \in \mathcal{F}$, is inbound to a distinct non-root vertex, $v \in \mathcal{V}$, (corollary 3.1.3) and since $(u \xrightarrow{T} v) \prec v$ (def. 3.1.4) and that $(u \xrightarrow{T} v)$ and v are contained in the same segment (def. 4.1.2) then there is always a tree edge with lower precedence contained in the same segment as each non-root vertex; therefore no non-root vertex can be at the head of a segment.

Cotree edges are leaves of the Trémaux Tree (def. 4.1.11) and are at the tail of their respective segments (lemma 4.1.12) so have the highest precedence within those segments. Therefore vertices or tree edges in a segment with a cotree edge must have lower precedence so a cotree edge cannot be at the head of a segment unless it is the only element in that segment.

Given this, for any segment: if it contains the root vertex then that will be at the head of the segment; if the segment contains vertices, but not the root vertex, then it also contains a distinct tree edge for each non-root vertex and the minimum precedence element is the minimum precedence tree edge; otherwise, the segment contains no vertices and, hence, no tree edges and so only a single cotree edge which is head (and tail) of the segment. $\square$

COROLLARY 4.1.17. For a non-root segment s then $\text{head}(s)$ is an edge $(u \rightarrow v)$. Since vertex u immediately precedes $(u \rightarrow v)$ (def. 3.1.4) therefore u is contained in the maximum precedence ancestor segment of s – the parent segment of s .

DEFINITION 4.1.18. A bridge edge is an edge of $\mathcal{G}$ not contained in any biconnected components of $\mathcal{G}$ (and, hence, is not contained in any cycles of $\mathcal{G}$).

LEMMA 4.1.19. Each cotree edge is part of a cycle and, hence, a biconnected component.

PROOF 4.1.20. Each cotree edge $(u \xrightarrow{C} v)$ has the relationship $v \preceq u \prec (u \xrightarrow{C} v)$ (def. 3.1.4) so the chain $[v; u]$ and $(u \xrightarrow{C} v)$ define a cycle; hence $(u \xrightarrow{C} v)$ is part of a biconnected component. $\square$

COROLLARY 4.1.21. All bridge edges are tree edges.

LEMMA 4.1.22. A given tree edge $(u \xrightarrow{T} v)$ is a bridge edge if, and only if, $\text{lowPoint1}(v) = v$.

PROOF 4.1.23. Given that $\text{lowPoints}(v) \subseteq [r; v]$ (def. 3.1.8) and that no vertex succeeds u and also precedes v (def. 3.1.4) then either $\text{lowPoint1}(v) = v$ or $\text{lowPoint1}(v) \preceq u$.

If $\text{lowPoint1}(v) \preceq u$ then, for that low point to be reachable, a cotree edge $(x \xrightarrow{C} \text{lowPoint1}(v))$ exists such that $\text{lowPoint1}(v) \preceq u \prec (u \xrightarrow{T} v) \prec v \preceq x \prec (x \xrightarrow{C} \text{lowPoint1}(v))$. Given this then the chain $[\text{lowPoint1}(v); (x \xrightarrow{C} \text{lowPoint1}(v))]$ defines a cycle and contains $(u \xrightarrow{T} v)$; therefore if $\text{lowPoint1}(v) \preceq u$ then $(u \xrightarrow{T} v)$ is contained in a cycle and is not a bridge edge.

If $\text{lowPoint1}(v) = v$ then all edges reachable from v (succeeding v) are inbound to and outbound from vertices succeeding or equalling v so no edges reachable from v connect to u or its predecessors; given this then $(u \xrightarrow{T} v)$ cannot be contained in a cycle and is a bridge edge.

Therefore $(u \xrightarrow{T} v)$ is a bridge edge if, and only if, $\text{lowPoint1}(v) = v$. $\square$

COROLLARY 4.1.24. Given a tree edge $(u \xrightarrow{T} v)$ where vertex v has no outbound edges then $\text{reachable}(v) = \{v\} = \text{lowPoints}(v)$ therefore $\text{lowPoint1}(v) = v$ so $(u \xrightarrow{T} v)$ is a bridge edge.

LEMMA 4.1.25. All edges which are contained in the same segment as, and precede, a bridge edge are also bridge edges.

PROOF 4.1.26.

- [1] $(v \xrightarrow{T} w)$ is a bridge edge $\Rightarrow \text{lowPoint1}(w) = w$ (lemma 4.1.22)
- $\Rightarrow \text{lowPoint1}((v \xrightarrow{T} w)) = v$ (def. 3.1.8)
- [2] $(u \xrightarrow{T} v)$ and $(v \xrightarrow{T} w)$ are in the same segment
- $\Rightarrow (v \xrightarrow{T} w) = \min^{**}(\text{outbound}(v))$ (def. 4.1.2)
- $\Rightarrow \forall e \in \text{outbound}(v) : \text{lowPoint1}((v \xrightarrow{T} w)) \preceq \text{lowPoint1}(e)$
- [1] and [2] $\Rightarrow \forall e \in \text{outbound}(v) : \text{lowPoint1}(e) = v$
- $\Rightarrow \text{lowPoint1}(v) = v$
- $\Rightarrow (u \xrightarrow{T} v)$ is a bridge edge (lemma 4.1.22)

Therefore, if tree edge $(v \xrightarrow{T} w)$ is a bridge edge and an immediately preceding tree edge $(u \xrightarrow{T} v)$ is contained in the same segment then it will also be a bridge edge. Applying this recursively to each preceding (tree) edge then all edges contained in the same segment as, and preceding, a given bridge edge are also bridge edges. $\square$

COROLLARY 4.1.27. If a segment's tail is a vertex (with no outbound edges) then, given lemma 4.1.25 and corollary 4.1.24, all edges in that segment are bridge edges.

LEMMA 4.1.28. Considering the tail vertex of a given segment then:

- All elements within that segment that equal or succeed its tail vertex have identical first low points; and
- All edges within that segment that precede its tail vertex are bridge edges and those bridge edges each have different first low points equal to the vertex immediately preceding that bridge edge.

PROOF 4.1.29. Given a segment containing vertex v and, hence, also containing the immediately preceding tree edge $(u \xrightarrow{T} v)$ (def. 4.1.2) then, considering the first low point of vertex v , either:

- $\text{lowPoint1}(v) \preceq u$, in which case:
 - $(u \xrightarrow{T} v)$ is not a bridge edge (lemma 4.1.22);
 - The minimum reachable vertex from $(u \xrightarrow{T} v)$ is either vertex u or the minimum reachable vertex from vertex v and since this precedes or equals u then $\text{lowPoint1}((u \xrightarrow{T} v)) = \text{lowPoint1}(v)$;
 - Therefore each non-bridge tree edge has the same first low point as the immediately succeeding vertex within the same segment; or
- $\text{lowPoint1}(v) = v$, in which case:
 - $(u \xrightarrow{T} v)$ is a bridge edge (lemma 4.1.22);
 - All other preceding edges in that segment are also bridge edges (lemma 4.1.25);
 - For each bridge edge $(u \xrightarrow{T} v)$, since $\text{lowPoint1}(v) = v$, then $\text{lowPoint1}((u \xrightarrow{T} v)) = u \neq \text{lowPoint1}(v)$;
 - Therefore each bridge edge has a first low point equal to the immediately preceding vertex and different to the immediately succeeding vertex so all bridge edges within a segment have different first low points.

Considering a segment such that edge $(x \rightarrow y)$ and the immediately preceding vertex x are both contained in that segment then: $\text{lowPoint1}((x \rightarrow y)) \preceq x$ (def. 3.1.8); and since $(x \rightarrow y)$ has the minimum TT precedence of all edges outbound from x (def. 4.1.2) then its first low point precedes or equals the first low point of all other edges outbound from x therefore, $\text{lowPoint1}(x) = \text{lowPoint1}((x \rightarrow y)) \preceq x$.

Given the above statements, considering a segment containing the elements $(u \xrightarrow{T} v)$, v and $(v \rightarrow w)$ then:

- If $\text{lowPoint1}((v \rightarrow w)) \preceq u$ then $\text{lowPoint1}((u \xrightarrow{T} v)) = \text{lowPoint1}(v) = \text{lowPoint1}((v \rightarrow w))$; and
- If $\text{lowPoint1}((v \rightarrow w)) = v$ then $(u \xrightarrow{T} v)$ is a bridge edge, $\text{lowPoint1}((u \xrightarrow{T} v)) = u$ and $\text{lowPoint1}(v) = \text{lowPoint1}((v \rightarrow w)) = v$.

Considering a segment s then either:

- The segment's tail is a vertex v with no outbound edges (fig. 4.1.2a). In which case, all edges in that segment are bridge edges (each with different first low points) and v is the only element equalling or succeeding the tail vertex (which is also v);
- The segment's tail is a cotree edge $(u \xrightarrow{C} v)$ where $v \preceq \text{head}(s)$ (fig. 4.1.2c or d) then, by recursively applying the above statement, the segment contains no bridge edges and all elements have identical first low points; or
- The segment's tail is a cotree edge $(u \xrightarrow{C} v)$ where $\text{head}(s) \prec v$ (fig. 4.1.2b) then:
 - By recursively applying the above statement, the elements within that segment succeeding or equal to v all have a first low point equal to v ; and
 - Given that $\text{head}(s) \prec v$ there must exist a tree edge $(x \xrightarrow{T} v)$ immediately preceding v and since $\text{lowPoint1}(v) = v$ then $(x \xrightarrow{T} v)$ is a bridge edge (and so are all other preceding edges within s).

Therefore, in all cases the lemma is true. □

4.1.1 Ordering the Segments

DEFINITION 4.1.30. A total order ($<^s$) can be defined on the set of segments such that $s_\alpha <^s s_\beta$ if:

- s_α is an ancestor of s_β ;
- s_α is a sibling of s_β and $\text{headVertex}(s_\beta) \prec \text{headVertex}(s_\alpha)$;
- s_α is a sibling of s_β , $\text{headVertex}(s_\alpha) = \text{headVertex}(s_\beta)$ and $\text{head}(s_\alpha) <^{**} \text{head}(s_\beta)$; or
- s_α is a descendant of or equal to s'_α and s_β is a descendant of or equal to s'_β such that s'_α is a sibling of s'_β and $s'_\alpha <^s s'_\beta$.

EXAMPLE 4.1.31. Given the segments from figure 4.1.1 then $[A; (L \xrightarrow{C} A)]$ is the root segment and the other segments descend from it as described in figure 4.1.4 (below) and are in $<^s$ precedence order as numbered in figure 4.1.3 (right).

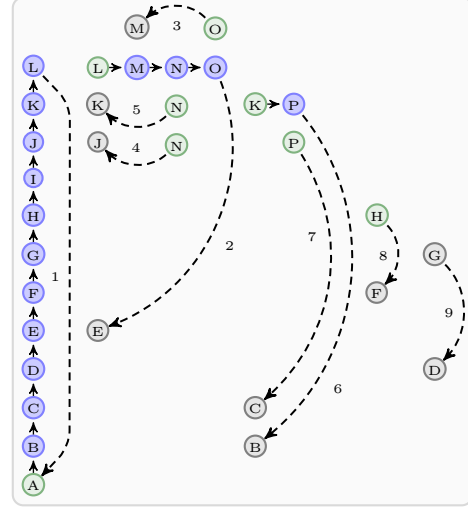


Figure 4.1.3: Segments from Figure 4.1.1
Numbered in $<^s$ Order

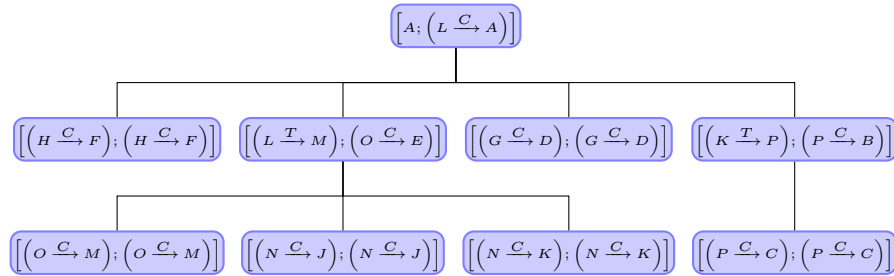


Figure 4.1.4: Segment Parent-Child Relationships from Figure 4.1.1

COROLLARY 4.1.32. Considering definition 4.1.30 and the precedence of the head vertices and head edges of related segments then:

- If segment s_j is a descendant of segment s_i then $s_i <^s s_j$ and $\text{headVertex}(s_i) \prec_{\text{im}} \text{headEdge}(s_i) \prec \text{headVertex}(s_j) \prec_{\text{im}} \text{headEdge}(s_j)$;
- If segment s_i is a sibling of segment s_j where $s_i <^s s_j$ then $\text{headVertex}(s_j) \preceq \text{headVertex}(s_i)$ and $\text{headEdge}(s_i) \not\prec \text{headEdge}(s_j)$ (since s_i and s_j are in different branches of the Trémaux Tree);
- If segment s_j is a descendant of a sibling segment s_i of segment s_k where $s_i <^s s_k$ then $s_i <^s s_j <^s s_k$ and $\text{headVertex}(s_k) \preceq \text{headVertex}(s_i) \prec_{\text{im}} \text{headEdge}(s_i) \prec \text{headVertex}(s_j) \prec_{\text{im}} \text{headEdge}(s_j)$ and $\text{headEdge}(s_i) \not\prec \text{headEdge}(s_k)$ and $\text{headEdge}(s_j) \not\prec \text{headEdge}(s_k)$.

4.1.2 Varying the Order of Incomparable TT-Precedence Edges

Definition 3.1.13 defines a total order ($<^{**}$) over each vertex's set of outbound edges by assigning an arbitrary order to those edges which have an incomparable TT-precedence ($\parallel^*$). Considering a set of $\parallel^*$ edges outbound from a given vertex v which is contained in a segment s then any given edge will be either:

- The minimal $<^{**}$ edge – in which case, it will be contained in s . Varying the arbitrary order assigned to that set of $\parallel^*$ edges such that a different edge has minimal precedence results in a different chain of elements within s , succeeding v but having identical tail vertex (since all $\parallel^*$ edges have the same first low point).
- A non-minimal $<^{**}$ edge – in which case, all these edges will be at the head of segments which are siblings and are children of s . By definition 4.1.30, the $<^{**}$ propagates through to the $<^s$ order for a segment's children; by varying the arbitrary $\parallel^*$ order then the $<^s$ order varies and, importantly, vice versa. When considering a subset of the total order $<^s$ corresponding to the set of children of a given segment, an additional point to note, is that child segments with $\parallel^*$ head edges will, by definition 4.1.30, be sequentially ordered within this subset of children.

The impact of varying the order of the segments, in the context of a planar embedding, is considered in the following section.

4.2 Planar Embedding

This section deals with a planar embedding Γ of a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ onto a surface $\mathcal{M}$ (which is assumed, in this and future references, to be orientable and of genus 0; i.e. the $\mathbb{R}^2$ plane or the surface of a sphere). It considers the incremental addition of segments, in $<^s$ precedence order, to the embedding such that each successive segment has either:

- A planar embedding directly within the embedding of lower $<^s$ precedence segments;
- A planar embedding within the embedding of lower $<^s$ precedence segments after they have been re-oriented via one or more *Whitney flips*; or
- No possible planar embedding even after considering re-orientation of previously embedded segments via *Whitney flips*, so the graph is non-planar.

In addition, the case is considered where all segments are embedded and re-orientation of the segments is still possible thus allowing for multiple embeddings of the same graph.

DEFINITION 4.2.1. A graph is defined to be *planar* in the following, equivalent, conditions:

- It has a planar embedding (a topological representation without edge crossings on an orientable surface of genus 0 – i.e. the surface of a sphere or the $\mathbb{R}^2$ plane);
- It does not contain a sub-graph homeomorphic to K_5 or $K_{3,3}$ [44];
- It does not contain K_5 or a $K_{3,3}$ as a minor [78];
- It has a dual [81]; and
- The Trémaux Tree of the graph does not contain a pair of cotree edges which are defined to be both *T-opposite* and *T-alike* [18].

4.2.1 Embeddings, Faces and Regions

DEFINITION 4.2.2. An embedding is a mapping of a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ onto a surface $\mathcal{M}$ such that each vertex maps to a point on $\mathcal{M}$ and each edge maps to an open curve (or closed curve for a looping edge) on $\mathcal{M}$ terminating at the points corresponding to the incident vertices. A planar embedding Γ is such that each vertex maps to a distinct point on $\mathcal{M}$ and each edge is a Jordan arc (or, for looping edges, a Jordan curve), which does not intersect with any points mapping to other edges or vertices except at its incident vertices.

Given a sequence of segments $\langle s_1 \dots s_\omega \rangle$ (in $<^s$ order) then a sequence of sub-graphs $\langle \mathcal{G}_0 \dots \mathcal{G}_\omega \rangle$ can be formed, where $\mathcal{G}_n = \bigcup_{1 \leq k \leq n} s_k$. If $\mathcal{G}_\omega$ is planar then each sub-graph is also planar and there exists a corresponding sequence $\langle \Gamma_0 \dots \Gamma_\omega \rangle$ of planar embeddings of those sub-graphs on the surface $\mathcal{M}$. The following terminology is used relating to those embeddings:

DEFINITION 4.2.3. A *face* of Γ_n is a maximal set of connected points within the complement $\mathcal{M} \setminus \Gamma_n$. A vertex/edge is defined to be *adjacent* to face (and vice versa) if that vertex/edge is contained in the boundary of that face; similarly, a face is *adjacent* to a segment if a vertex or edge contained in that segment is *adjacent* to that face. Thus, a *face* of Γ_n may, in Γ_x (where $n < x$), contain the embedding of segment s_x and so have been sub-divided and not exist in that later embedding; this means that the number of *faces* adjacent to a vertex/segment can increase with successive embeddings.

DEFINITION 4.2.4. A *region* relative to segment s_n is defined to be a face of Γ_n adjacent to s_n at the instant of its embedding.

COROLLARY 4.2.5. Given that each segment s_n is a chain of connected elements then a planar embedding of s_n must be entirely within a face f of $\Gamma_{(n-1)}$ and so the *region(s)* relative to s_n correspond to the face(s) formed by the subdivision of f by s_n and all elements of s_n are, at that instant, only adjacent to those face(s) composing the *region(s)* relative to that segment.

Faces are subdivided by subsequent embeddings and change with those sub-divisions, however, regions are constant areas and are comprised of one or more faces and zero or more (descendant) segments and can contain nested regions corresponding to the segment hierarchy.

LEMMA 4.2.6. If a graph has a planar embedding on the surface of a sphere then there is a corresponding planar embedding on the $\mathbb{R}^2$ plane, and vice versa.

PROOF 4.2.7. The surface of a sphere can be mapped, using a stereographic projection¹ (figure 4.2.1), onto a $\mathbb{R}^2$ plane; thus by choosing a point, on the surface of the sphere, within a face of the embedding as the focal point of the projection then a stereographic projection maps the embedding to a corresponding embedding in the plane. The inverse projection maps an embedding in the $\mathbb{R}^2$ plane onto the surface of a sphere. $\square$

The implication of Lemma 4.2.6 is that a graph's embedding can be, equivalently, considered on the surface of a sphere and in the $\mathbb{R}^2$ plane.

4.2.2 Segment Classes

Recalling definition 4.1.14, this details four classes of segment (examples of which are given in figure 4.2.2) based on precedence of the tail vertex.

NOTATION 4.2.8. Defining a shorthand notation for the head, tail and low points of a segment s_x :

- $v_h^x = \text{headVertex}(s_x)$,
- $v_t^x = \text{tailVertex}(s_x)$,
- $e_h^x = \text{headEdge}(s_x)$,
- $e_t^x = \text{tailEdge}(s_x)$,
- $v_{l_1}^x = \text{lowPoint1}(e_h^x)$ and
- $v_{l_2}^x = \text{lowPoint2}(e_h^x)$.

Using this, a segment s_x of a given Class has the following relationships between elements:

Class 1: $v_{l_1}^x = v_{l_2}^x = v_h^x \prec_{\text{IM}} e_h^x \preceq e_t^x \prec_{\text{IM}} v_t^x$;

Class 2: $v_{l_1}^x = v_{l_2}^x = v_h^x \prec_{\text{IM}} e_h^x \prec v_t^x \prec e_t^x$;

Class 3: $v_{l_1}^x = v_{l_2}^x = v_h^x = v_t^x \prec_{\text{im}} e_h^x \preccurlyeq e_t^x$; and

Class 4: $v_{l_1}^x = v_t^x \prec v_{l_2}^x \preccurlyeq v_h^x \prec_{\text{IM}} e_h^x \preccurlyeq e_t^x$.

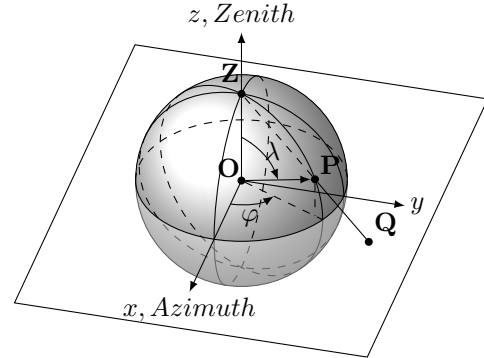


Figure 4.2.1: Stereographic Projection Mapping Between the Surface of a Unit Sphere ($P = (1, \lambda, \varphi)$) and the $\mathbb{R}^2$ Plane ($Q = (\cot(\lambda/2), \varphi)$)²

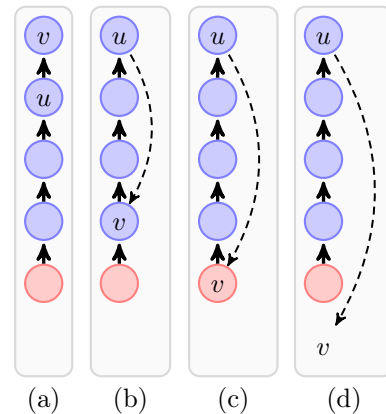


Figure 4.2.2: Examples of Segment Classes Based on Tail Vertex Precedence (Head Vertex in Red)

¹ The $\mathbb{R}^2$ reference plane can be described using polar co-ordinates $Q = (r, \theta)$, relative to the origin O . The surface of a unit sphere can be described using spherical co-ordinates $P = (1, \lambda, \varphi)$ where for given zenith (normal to the reference plane) and azimuth (within the reference plane) reference directions, emanating from the origin O , then:

- The intersection of the unit sphere and the zenith reference direction is at $Z = (1, 0, \varphi_z)$ where φ_z is arbitrary but, by convention, can be defined to be 0;
- φ is the azimuth angle (the angle $[0; 2\pi[$ between the azimuth reference direction and the orthogonal projection of the line OP onto the reference plane);
- λ is the zenith angle (the angle $[0; \pi]$ between the zenith reference direction and OP).

A stereographic projection, using Z as the projection's focal point, maps P to Q where Q is the point of intersection of the reference plane and line through ZP (such that $\theta = \varphi$ and $r = \cot(\lambda/2)$).

² L^AT_EX PGF/Tikz Stereographic Projection code adapted from Tomek M. Trzeciak's code from <http://www.latex-community.org/forum/viewtopic.php?f=4&t=2111&p=8260>

4.2.3 Defining Segments As “Having An Embedding”

Each segment is: a simple path (Class 1 or 4); a simple cycle (Class 3); or a simple cycle with a branching path connected at a single vertex (Class 2) – so, when considered independently, each has a planar embedding within a surface. Given this then the construction of a planar embedding can be considered such that:

- The root segment s_1 always has a planar embedding Γ_1 within an empty surface Γ_0 ; and
- Each subsequent segment s_k has a planar embedding Γ_k within the planar embedding Γ_{k-1} of all previously embedded segments $\{s_1, s_2, \dots, s_{k-1}\}$ (all having lower $<^s$ precedence than s_k) if, and only if, there exists a face of Γ_{k-1} whose boundary contains all vertices incident to that segment (i.e. its head vertex and, for a Class 4 segment, its tail vertex) and, hence, that segment can be embedded without crossing the boundary to that face.

To prevent a segment being considered for embedding in a face that cannot contain an embedding of that segment’s descendants then this definition is further restricted such that:

DEFINITION 4.2.9. A non-root segment s_x is defined as having an *embedding* if, within a planar embedding of the sub-graph formed exclusively of all lower $<^s$ precedence segments, there exists a face f with cycle $\mathcal{C}_f$ of boundary vertices where $(\{v_{l_1}^x\} \cup [v_{l_2}^x; v_h^x]) \subseteq \mathcal{C}_f$.

COROLLARY 4.2.10. Any face that can contain an embedding of a segment contains all low points of that segment on its boundary; this follows directly from definition 4.2.9, since $\text{lowPoints}(e_h^x) \subseteq (\{v_{l_1}^x\} \cup [v_{l_2}^x; v_h^x])$. Therefore, all vertices at which a segment’s descendants connect to an ancestor of that segment are also contained on the boundary of the face containing the embedding of that segment.

COROLLARY 4.2.11. As a corollary to this and definition 4.2.4, each segment’s embedding is either a simple path (Class 1) or defines a cycle, either of itself (Class 2 or 3) or with ancestors (Class 4). By Euler’s formula [22] and the Jordan Curve Theorem [39] then: the embedding of a Class 1 segment defines one adjacent region; and the embedding of a Class 2, 3 or 4 segment defines two adjacent regions which can be designated as the “inside” and “outside” regions relative to that segment.

Considering this further:

THEOREM 4.2.12. If the biconnected components of a graph $\mathcal{G}$ are planar, $\mathcal{G}$ is planar.

PROOF 4.2.13. This is proved by Whitney [81, Pg. 356]: “Suppose the graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ are planar, and $\mathcal{G}'$ is formed by letting the vertices a_1 and a_2 of $\mathcal{G}_1$ and $\mathcal{G}_2$ [respectively] coalesce. [...] Map $\mathcal{G}_1$ on a sphere, and map $\mathcal{G}_2$ on a plane so that one of the regions adjacent to the vertex a_2 is the outside region. Shrink the portion of the plane containing $\mathcal{G}_2$ so it will fit into one of the regions of $\mathcal{G}_1$ adjacent to a_1 . Drawing a_1 and a_2 together, we have mapped $\mathcal{G}'$ on the sphere. The theorem follows as a repeated application of this process.” $\square$

COROLLARY 4.2.14. Since each non-root Class 1, 2 and 3 segment defines a 1-separation at its head vertex³ (separating that segment and its descendants from its ancestors and those ancestors other descendants) then appending a Class 1, 2 or 3 segment within a face of planar embedding always results in a planar embedding.

COROLLARY 4.2.15. Following corollary 4.2.14, a planar embedding of a graph can be constructed segment-by-segment (in ascending $<^s$ order) if, and only if, each Class 4 segment has a planar embedding within a face of the (respective biconnected component of the) sub-graph formed by all lower $<^s$ segments.

³ And, given that $\text{lowPoints}(\text{head}(s_i)) = \{\text{headVertex}(s_i)\}$ when s_i is of Class 1, 2 or 3, then this property is maintained as subsequent segments are appended to the embedding.

4.2.4 Defining Sides to the Root Segment

When considering the embedding of the root segment in the $\mathbb{R}^2$ plane, convention describes the “inside” (“outside”) of a Jordan curve to be the finite (infinite) bounded adjacent faces; mapping this embedding to a sphere (via stereographic projection) then rotating (or mirroring) to swap the northern and southern hemispheres before mapping back to the $\mathbb{R}^2$ plane results in reversal of the “inside” and “outside” faces and the natural affordance between the $\mathbb{R}^2$ surface and the designation of these faces is lost. Therefore, it is useful to designate a segment’s adjacent faces/sides as “inside” or “outside” independent of any presumptions of the (orientable genus 0) surface being used for the embedding; this can be achieved by attaching a child segment to one side of the root and defining that side to be the “inside” and the other side to be the “outside”.

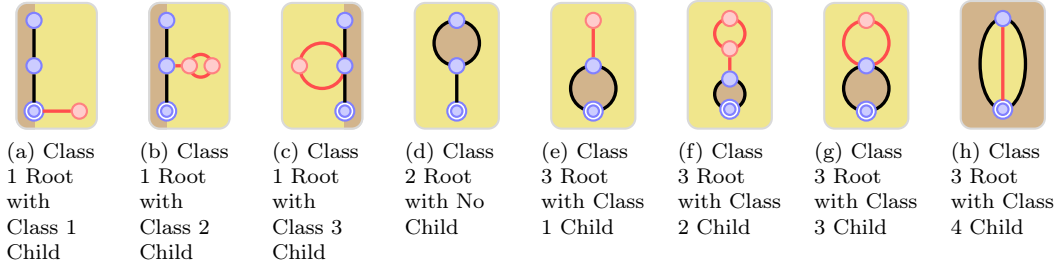


Figure 4.2.3: Inside (Yellow) and Outside (Brown) Adjacent Regions of Different Classes of Root Segment (Black/Blue) as Defined by Location of Branching Path/Cycle

Figure 4.2.3 shows minimal examples of the possible classes of root segment and children (if any) that are required to define the sides adjacent to the root segment.

- A Class 1 root segment (figures 4.2.3a-c) requires a child segment to define a side. One can consider extending the path defined by a Class 1 root segment to form a cycle (for example: by including the root segment as part of the boundary of a cycle with infinite radius in the $\mathbb{R}^2$ plane; or as a section of a great circle on the surface of a sphere) to separate the surface into two regions – positioning the child segment within one of these regions defines that region to be the “inside” and the opposing region as the “outside”.
- A Class 2 root segment (figure 4.2.3d) requires no child segment as the side of the cycle containing the leading path of bridge edges can be defined to be the “inside” and the opposite side of the cycle as the “outside”.
- A Class 3 root segment (figures 4.2.3e-h) requires a child segment to define a side such that the side of the root segment containing that child is defined as the “inside” and the opposite side of the cycle as the “outside”.

This leads to the definition of a *static* segment:

DEFINITION 4.2.16. The root segment is defined to have a *static* embedding (or, simply, to be *static*) such that the root segment defines a cycle (either of itself, for a Class 2 or 3 root segment, or by extension, for a Class 1 root segment) and that the cyclic orientation of the root segment’s edges about that cycle is in a *static* direction. A branching path/cycle (a leading path of bridge edges, for a Class 2 segment, or by the root segment’s child with minimal $<^s$ precedence, for a Class 1 or 3 segment) within one region bounded by that cycle defines that region to be the *inside* and the opposing region to be the *outside*.

This is *static* since, regardless of transformations to the embedding of subsequent segments, the root segment’s cyclic direction and the branching path/cycle’s containing region is unchanged⁴.

COROLLARY 4.2.17. Since no sub-graph formed by the root segment and any *static* branching path/cycle is homeomorphic to either a K_5 or a $K_{3,3}$ then the *static* segment(s) must have a planar embedding.

⁴ Although the cyclic direction of a branching cycle can be changed within that region

4.2.5 Defining the Inside and Outside Relative to Non-Root Segments

Considering the non-root segments then it is useful to extend the definition of “inside” and “outside” used for the root segment:

DEFINITION 4.2.18. Given a planar embedding Γ_x of a segment s_x which forms a cycle, either of itself (Class 2 or 3) or with its ancestors (Class 4), within a face f (with boundary $\mathcal{C}_f$) of the embedding Γ_{x-1} then, by the Jordan Curve Theorem (see page 9), the segment bisects f forming two regions/faces adjacent to, and separated by, that segment. These regions can be designated as *inside* and *outside* of s_x such that:

- If s_x is a Class 2 or 3 segment then the inside region is bounded by the cyclic chain $[v_t^x; e_t^x]$ and the outside region is bounded by $(\mathcal{C}_f \cup s_x)$; and
- If s_x is a Class 4 segment then $\mathcal{C}$ is a Jordan curve and can be partitioned, at v_h^x and v_t^x , to form two Jordan arcs $\mathcal{C}_1$ and $\mathcal{C}_2$ where: $\mathcal{C}_2 = (\mathcal{C}_f \setminus \mathcal{C}_1) \cup \{v_h^x, v_t^x\}$; and $\mathcal{C}_1$ ($\mathcal{C}_2$) is defined to be the arc containing the edge immediately preceding/inbound to (not preceding/outbound from) v_h^x . Given these definitions then the inside region is defined to be the face bounded by $(\mathcal{C}_1 \cup s_x)$ and the outside region is defined to be the face bounded by $(\mathcal{C}_2 \cup s_x)$.

COROLLARY 4.2.19. For a Class 4 segment s_x to have an embedding within face f , then $\mathcal{C}_f$ contains $v_{l_1}^x$ and the chain $[v_{l_2}^x; v_h^x]$ (definition 4.2.9). Given this (and that $v_{l_1}^x \prec v_h^x$ so there is always an edge immediately preceding v_h^x), then considering the regions formed:

- The inside region contains the immediately preceding edge to v_h^x within $\mathcal{C}_f$, so is bounded (in part) by the path $([v_{l_2}^x; v_h^x] \cup s_x \cup \langle v_t^x \rangle)$; and
- The outside region contains the immediately succeeding edge to v_h^x within $\mathcal{C}_f$, so is bounded (in part) by the path $(\langle v_h^x \rangle \cup s_x \cup \langle v_t^x \rangle)$ and the chain $[v_{l_2}^x; v_h^x]$ does not bound the outside region.

Figure 4.2.4, below, gives examples of the inside and outside regions formed when a cyclic segment is embedded within a face. Figure 4.2.4a is an example of the embedding of a Class 2 segment; figure 4.2.4b is a similar example for a Class 4 segment; and figure 4.2.4c extends figure 4.2.4b by embedding an additional Class 4 segment within the previously generated outside face.

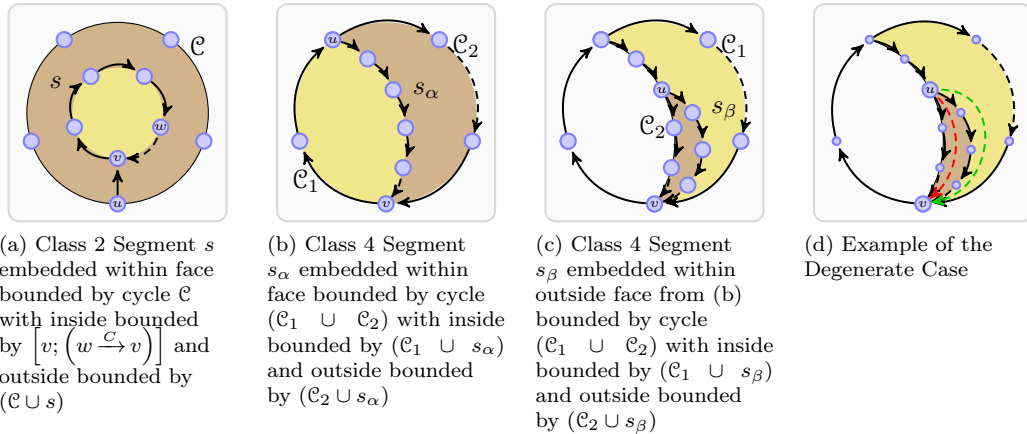


Figure 4.2.4: Examples of Inside (Yellow) and Outside (Brown) Regions Formed by Embedding Different Segment Classes

A degenerate case (figure 4.2.4d – red embedding) exists when the boundary of the (brown) face contains two edges immediately succeeding and no edge immediately preceding the head vertex. This case occurs when two segments have $\parallel^*$ precedence head edges (where both segments have a common head and second low point vertex) and is considered in more depth on pages 51 & 68; however, the $<^s$ embedding order ensures there is always an embedding (green) in a (yellow)

face containing edges that immediately precede and succeed the head vertex so the segment can be embedded into that face and by carefully permuting the order the segments are embedded (which, in this situation, is permissible since the $<^s$ order of this pair of segments is based on an arbitrary ordering of the $\parallel^*$ head edges) then the degenerate case can be formed whilst maintaining the property that a segment is always embedded within a face containing an edge inbound to that segment's head.

4.2.6 Embedding Class 1, 2 or 3 Segments

Considering definition 4.2.9, we recall that given the embedding of a segment s_x into a face f (which is bounded by the cycle of vertices $\mathcal{C}_f$) then $\text{lowPoints}(e_h^x) \subseteq \mathcal{C}_f$.

If s_x is a class 1, 2 or 3 segment then it connects to $\mathcal{C}_f$ at a single vertex (its head vertex) and, by corollary 4.2.15, always has a planar embedding. Given that any face containing that head vertex can contain that embedding then this can be further expanded such that a Class 1, 2 or 3 segment always has an embedding within any face adjacent to its head vertex (figure 4.2.5) and as such:

DEFINITION 4.2.20. All non-*static* Class 1, 2 or 3 segments are defined to be *rotatable* (in that the embedding of that segment can be rotated around its head vertex and contained within any adjacent face).

Further to this, each Class 2 or 3 segment defines a cycle and that cycle can be embedded such that the edges bounding that cycle have either a clockwise or anti-clockwise direction about the inside face of the embedding (figure 4.2.6). The root segment is fixed to have a *static* cyclic embedding direction but all other Class 2 or 3 segments can have the cyclic direction of their embedding reversed and as such:

DEFINITION 4.2.21. All non-root Class 2 or 3 segments are defined to be *reversible* (in that the cyclic direction of its edges, around the inside face defined by its embedding, can be reversed).

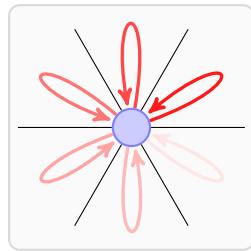


Figure 4.2.5: Example of Possible Embeddings of (Red) Segment Rotating About Head Vertex Into Different Adjacent Faces

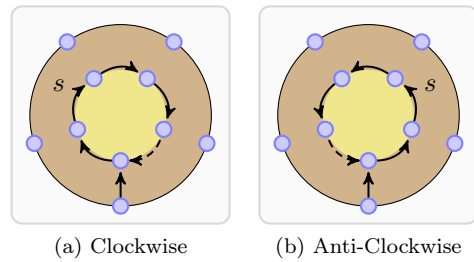


Figure 4.2.6: Example Clockwise Embedding of Class 2 Segment s and Reverse Anti-Clockwise Embedding

4.2.7 Embedding Class 4 Segments

Each non-root segment connects to its head vertex which, by definition, is contained in its parent segment. That parent's embedding forms either: one adjacent face, if the parent is of Class 1; or, by the Jordan Curve Theorem, two adjacent faces (inside and outside) separated by that parent segment, if that parent is of Class 2, 3 or 4. Given this then, if a non-root segment has a planar embedding it must be within a face adjacent to its parent or a sub-division thereof formed by previously embedding sibling segments, or sibling's descendant segments (with lower $<^s$ precedence).

LEMMA 4.2.22. Given a segment currently being embedded then all of its previously embedded siblings and their descendants will have head vertices succeed or equal the head vertex of that given segment.

PROOF 4.2.23. Given the segments s_x , s_y and s_z where $s_x <^s s_z$ such that s_x and s_z are siblings then, by the $<^s$ ordering, $v_h^z \preceq v_h^x$; if s_y is a descendant of s_x then, by the $<^s$ ordering, $s_x <^s s_y <^s s_z$ so, by the parent-child relationship, $v_h^x \prec v_h^y$ and hence $v_h^z \preceq v_h^x \prec v_h^y$. $\square$

Considering sequence $\langle s_1, s_2, \dots, s_n \rangle$ of segments in ascending $<^s$ precedence order then a Class 4 segment s_k (where $v_t^k \prec v_h^k$) can be considered in the following cases:

1. When s_k has no siblings with lower $<^s$ precedence and, hence, s_{k-1} is the parent of s_k and $v_h^{k-1} \prec v_h^k$; or
2. When s_{k-1} is a sibling, or a sibling's descendant, of s_k and, hence, $v_h^k \preceq v_h^{k-1}$, which can be considered in the following sub-cases:
 - (a) $v_t^k \prec v_h^k \preceq v_{l_1}^{k-1} \preceq v_h^{k-1}$;
 - (b) $v_t^k \prec v_t^{k-1} \prec v_h^k \prec v_h^{k-1}$; or ⁵
 - (c) $v_t^{k-1} \preceq v_t^k \prec v_h^k \preceq v_h^{k-1}$, which can be separated into the further sub-cases:
 - i. $v_t^{k-1} \preceq v_t^k \prec v_h^k \prec v_h^{k-1}$;
 - ii. $v_t^{k-1} = v_t^k \prec v_h^k = v_h^{k-1}$; or
 - iii. $v_t^{k-1} \prec v_t^k \prec v_h^k \preceq v_h^{k-1}$.

Embedding a Class 4 Segment onto its Parent without Siblings

Considering case 1 then the parent s_{k-1} cannot be of Class 1 and so its embedding forms a cycle, either of itself (Class 2 or 3) or with ancestors (Class 4), defining an inside and an outside adjacent region. Considering the Class of the parent then it must either be:

- A Class 2 or 3 segment, which defines a cycle $[v_t^{k-1}; e_t^{k-1}]$ onto which s_k is connected. By the definitions of the segment classes and from the parent-child relationship, then $v_h^{k-1} \preceq v_t^{k-1} \preceq v_{l_1}^k = v_t^k \prec v_h^k \prec e_t^{k-1}$ hence $\text{lowPoints}(e_h^k) \subseteq [v_t^k; v_h^k] \subseteq [v_h^{k-1}; e_t^{k-1}]$ so all low point vertices of s_k are contained within the cycle defined by the parent s_{k-1} ; therefore, for a Class 2 or 3 parent segment:
 - s_k always has an embedding within either the inside and the outside face adjacent to s_{k-1} (provided s_k is not otherwise restricted by being defined to have a *static* embedding); and
 - There is always a pair of edges on that cycle which immediately precedes and immediately succeeds v_h^k (since $v_t^k \prec v_h^k \prec e_t^{k-1}$).

⁵ If $v_t^k \prec v_t^{k-1} \prec v_h^k = v_h^{k-1}$ then $s_k <^s s_{k-1}$ but since $s_{k-1} <^s s_k$ then this case cannot occur.

- A Class 4 segment, previously embedded into a face containing s_{k-1} 's low points ($\text{lowPoints}(e_h^{k-1})$) on its boundary. The inside and outside faces defined by the embedding of s_{k-1} are bounded, in part, by the paths $([v_{l_2}^{k-1}; v_h^{k-1}] \cup s_{k-1} \cup \langle v_t^{k-1} \rangle)$ and $(\langle v_h^{k-1} \rangle \cup s_{k-1} \cup \langle v_t^{k-1} \rangle)$, respectively (corollary 4.2.19).

If a vertex is reachable (definition 3.1.7) from e_h^k and given that $e_h^{k-1} \prec e_h^k$ then that vertex is also reachable from e_h^{k-1} therefore $\text{lowPoints}(e_h^k) \subseteq (\text{lowPoints}(e_h^{k-1}) \cup [e_h^{k-1}; v_h^k])$. Therefore, for a Class 4 parent segment:

- s_k can always be embedded within the inside face relative to its parent;
- If either $v_h^{k-1} \preceq v_t^k$ or $(v_t^{k-1} = v_t^k) \wedge (v_h^{k-1} \preceq v_{l_2}^k)$ then s_k can also be embedded within the outside face relative to its parent (provided s_k is not otherwise restricted by being defined to have a *static* embedding); and
- There is always a pair of edges on the path formed by s_{k-1} which immediately precedes and immediately succeeds v_h^k (since $v_h^{k-1} \prec v_h^k \prec e_t^{k-1}$).

This results in:

THEOREM 4.2.24. Any Class 4 segment s_c can always be embedded within the inside face defined by the embedding of its parent s_p . □

PROOF 4.2.25. Contained in the paragraphs above. □

THEOREM 4.2.26. Any Class 4 segment s_c can be embedded within the outside face defined by the embedding of its parent s_p if, and only if, $v_h^p \preceq v_t^c$ or $(v_t^p = v_t^c) \wedge (v_h^p \preceq v_{l_2}^c)$ (or, equivalently, if s_c has no low point vertices that are not on the boundary of the outside face defined by its parent – which will be any low point vertices that precede that parent's head vertex and succeed that parent's tail vertex) and that s_c is not restricted from being embedded in the outside face by being defined to have a *static* embedding. □

PROOF 4.2.27. Contained in the paragraphs above. □

COROLLARY 4.2.28. If the parent s_p is of Class 2 or 3 then $v_h^p \preceq v_t^p$ and, hence, no low point vertex v of a child s_c can exist such that $v_t^p \prec v \prec v_h^p$ so all children of a Class 2 or 3 segment can always be embedded within both the inside and outside face defined by the embedding of s_p (providing s_c is not *static*). □

Theorems 4.2.24 and 4.2.26 both lead to the definition that:

DEFINITION 4.2.29. A Class 4 segment s is defined to be *flippable* if s can be embedded within both the inside and the outside relative to its parent when those regions are considered devoid of any sub-divisions due to previously embedded siblings or sibling's descendants. □

This terminology is used as it represents the fact that a *flippable* segment s_c represents the potential existence of a *Whitney flip*, such that:

- Considering s_c 's parent (s_p) and the boundaries of the inside ($\mathcal{C}_I$) and outside ($\mathcal{C}_O$) regions adjacent to s_p then if s_c is flippable then the low points of s_c (including its head and, if Class 3 or 4, tail vertices) must be contained in both $\mathcal{C}_I$ and $\mathcal{C}_O$. $\mathcal{C}_I \cap \mathcal{C}_O = \{v_h^p, v_t^p\} \cup s_p$ therefore if s_c is flippable then $\text{lowPoints}(s_c) \in (\{v_h^p, v_t^p\} \cup s_p)$.
- If s_c is embeddable within a region adjacent to s_p then it is also embeddable within the region containing s_p adjacent to each ancestor of s_p since the regions adjacent to s_p are recursively nested within regions adjacent to each successive ancestor; therefore, no ancestor if s_p influences the embedding of s_c .
- A similar determination can be made for segments which are unrelated to s_p by noting that the head of those unrelated segments cannot be contained in the boundary $\mathcal{C}_I$ and since s_p has a valid embedding then no other vertex contained in, or incident to, any unrelated segment can be incident to s_c and either adjacent to the outside region relative to s_p or is not contained in any region relative to s_p so has no bearing on the embedding of s_c .

- By the definition of low points (3.1.8) then any child of s_c will have low points contained in $(\text{lowPoints}(s_c) \cup s_c)$ and so is also contained in either $\mathcal{C}_I \cap \mathcal{C}_O$ or s_c and does not restrict s_c from being embedded in either region. This point can be applied recursively to consider all descendants of s_c and thus if s_c is flippable then no descendant of s_c influences this (beyond the consideration of s_c 's low points).
- Since all other segments are accounted for then the only segments which can impact whether s_c can be embedded in both regions adjacent to its parent are s_c 's siblings and their descendants. Since these are specifically discounted by the definition of a segment being *flippable* then the *flippable* nature of a segment can be solely determined by comparing the precedence of $v_{l_1}^c$ and $v_{l_2}^c$ to v_h^p and v_t^p (such that if s_c is flippable then either $v_h^p \preceq v_{l_1}^c$ or $(v_t^p = v_{l_1}^c \wedge v_h^p \preceq v_{l_1}^c)$).

Thus if s_c is flippable then, unless restricted by a sibling or sibling's descendant, there exists a *Whitney flip* of the cycle defined by s_c and its parent. If a segment is not *flippable* then it is immediately restricted from being in the outside face so has a fixed embedding within the inside face (and a 3-separable minor exists formed from that segment and sufficient ancestors).

Embedding a Class 4 Segment onto its Parent with Siblings

Considering Case 2 (from page 46). Substituting s_j for s_{k-1} and assuming that s_j and s_k are within the same face adjacent to their common ancestor s_i , then the sub-cases take the forms shown in figure 4.2.7.

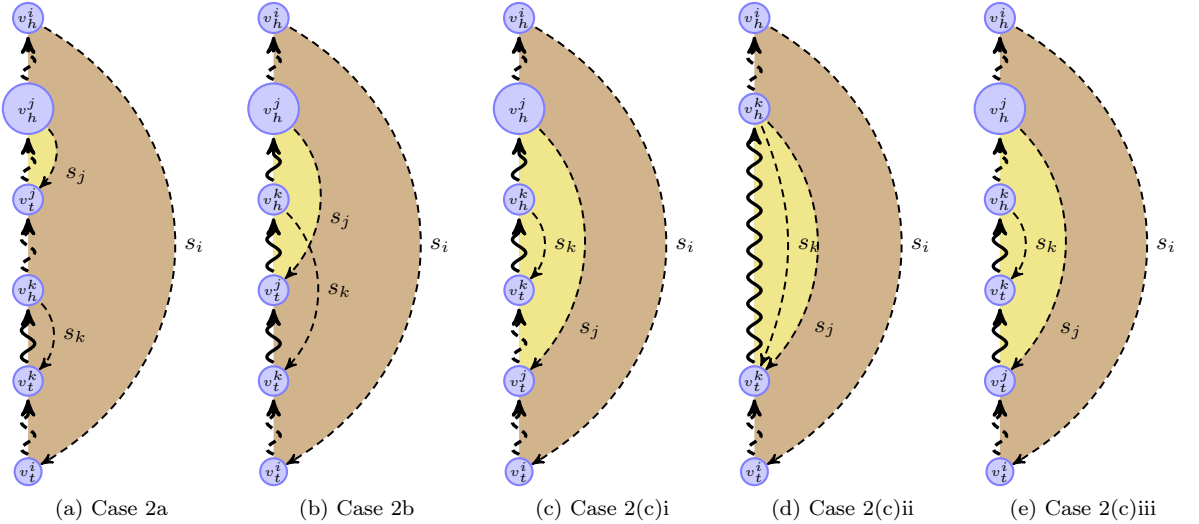


Figure 4.2.7: Embeddings of Segments s_i , s_j & s_k Where $s_i <^s s_j <^s s_k$ and s_i is the Parent of s_k and is an Ancestor of s_j (Sub-Cases of Case 2, Pg. 46) and Showing Inside (Yellow) and Outside(Brown) Faces Defined by s_j .

Partially and Fully Embedding Segments

Considering Case 2a (figure 4.2.7a) where $v_h^k \preceq v_{l_1}^j$ so s_k is contained within the outside face relative to s_j (and conversely, if s_k has an embedding then s_j is on the boundary of the outside face relative to s_k):

THEOREM 4.2.30. Given segments s_j and s_k where $s_j <^s s_k$ and $v_h^k \preceq v_{l_1}^j$ then no segment with higher $<^s$ precedence than s_k connects to a vertex with higher precedence than $v_{l_1}^j$.

PROOF 4.2.31. A segment s_x with higher $<^s$ precedence than s_k is (by definition 4.1.30) either:

- A descendant of s_k ;
- A sibling of s_k where $v_h^x \preceq v_h^k$;
- A descendant of a sibling of s_k where that sibling's head vertex precedes or equals v_h^k ;
- A sibling of an ancestor of s_k where $v_h^x \preceq v_h^k$; or
- A descendant of a sibling of an ancestor of s_k where that sibling's head vertex precedes or equals that ancestor's head vertex which precedes v_h^k .

Letting P be $[r; e_t^j]$, the path from the root vertex containing s_j .

If s_x is a sibling of s_k with higher $<^s$ precedence than s_k then (by definition 4.1.30) $v_h^x \preceq v_h^k$. Since each edge connects two comparable ($\perp$) precedence vertices (definition 3.1.1) and s_x branches at v_h^x from P then no vertex on P succeeding v_h^x is reachable from any element of s_x .

Similarly, if s_x is a sibling of an ancestor segment s_a of s_k where $s_k <^s s_x$ then $v_h^x \preceq v_h^a \prec v_h^k$ and s_x branches from P at v_h^x so no vertex on P succeeding v_h^x is reachable from any element of s_x .

Considering the cases when s_x is a descendant of, or a descendant of a sibling of, or a descendant of a sibling of an ancestor of, s_k then in all cases s_x is a descendant of a segment which branches from P and since the head vertex of that branching segment has the highest precedence of vertices on P that is reachable from any element of that branching segment then, by definitions 3.1.1 & 3.1.7, it follows that no descendant of that branching segment can reach a higher precedence vertex. Given that $v_h^k \preceq v_{l_1}^j$ and that the branching segments are either s_k , a sibling with a head vertex preceding or equalling s_k or an ancestor of s_k (which must have a head vertex preceding s_k) then no vertex is reachable that succeeds $v_{l_1}^j$.

Therefore, for any segment s_x where $s_j <^s s_k <^s s_x$ and $v_h^k \prec v_{l_1}^j$ then the maximum reachable vertex v from s_x on the path $[r; e_t^j]$ is such that $v \preceq v_h^k \preceq v_{l_1}^j$ and so s_x never connects to a vertex with higher precedence than $v_{l_1}^j$. $\square$

COROLLARY 4.2.32. Following directly from Theorem 4.2.30: Given two segments s_j and s_k where $s_j <^s s_k$ and where s_k is the minimal $<^s$ precedence segment such that $v_h^k \preceq v_{l_1}^j$ then s_k and all higher $<^s$ segments cannot be attached at two distinct vertices to the cycle defined by the embedding of s_j (since $v_{l_1}^j$ is the minimal precedence vertex within the cycle defined by the embedding of s_j).

This leads to two sub-categories of embedded segments:

DEFINITION 4.2.33. An embedded Class 4 segment s_j is categorised as being either:

- *partially embedded* if not all segments have been embedded and if no segment s_k has been embedded such that $s_j <^s s_k$ and $v_h^k \preceq v_{l_1}^j$; or
- *fully embedded* if either all segments have been embedded or if a segment s_k has been embedded such that $s_j <^s s_k$ and $v_h^k \preceq v_{l_1}^j$.

The definition of a *fully embedded* segment is extended on page 53 for simplicity of notation but, in its current form, is particularly useful when considering the successive segments as they are embedded since a *fully embedded* segment can neither define a state where non-planarity occurs nor have a Class 4 segment attached to the cycle it defines; therefore *fully embedded* segments are inconsequential to the embedding of future segments and so only *partially embedded* segments need be considered.

Planarity Conflicts

Considering Case 2b (page 46) – if s_k is to be embedded within the same face adjacent to its parent that contains s_j and where $v_t^k \prec v_t^j \prec v_h^k \prec v_h^j$ (hence, s_j is *partially embedded*) then s_j sub-divides that face such that s_k 's head and tail vertex are, respectively, in the inside and outside face defined by s_j 's embedding (and neither is on the boundary of that sub-division). Given this, then s_k does not have a planar embedding within the same face as s_j and, hence, if s_k has a planar embedding then it must be in the opposite face relative to s_k 's parent.

This leads to the definition:

DEFINITION 4.2.34. Given segments s_i , s_j and s_k where $s_i <^s s_j <^s s_k$ and s_i is an ancestor of, and the parent of, s_j and s_k , respectively, then if $v_t^k \prec v_t^j \prec v_h^k$ then s_j and s_k are defined to *conflict* and cannot be embedded within the same face relative to s_i .

Blocks of Conflicting Segments

It is useful to define a structure to represent groups of segments which *conflict* and must be embedded in opposite faces adjacent to their parent; to do this we introduce the concept of a *block* of segments:

DEFINITION 4.2.35. A block of segments (or, simply, a *block*) is defined relative to a given segment and is a group of Class 4 segments which are *partially embedded* children of that given segment (and their *partially embedded* descendants) where each of those segments is assigned to be embedded within either the inside or outside region relative to that given segment such that specifying the region containing the embedding of any one segment in that *block* defines which region every other segment of that *block* is embedded within.

COROLLARY 4.2.36. If a block contains more than one segment then every segment in that block must conflict with (and so be embedded in the opposite region to) at least one other segment in that block (and that each pair of segments within the block are connected by a series of *conflicts* and so: if there are an even number of conflicts in the series then the pair of segments must be embedded within the same region; or, if there are an odd number of conflicts then the pair of segments they must be embedded in opposing regions).

COROLLARY 4.2.37. If a segment conflicts with segments on both the inside and outside of a block then it has no planar embedding.

Any non-*flippable* segment added to a block must be embedded within the inside face relative that block's ancestor and so all segments within that block will then have a fixed embedding relative to that ancestor. This leads to an extension of the definition of segments being *flippable* to blocks such that:

DEFINITION 4.2.38. A block (relative to a given segment) is defined as being *flippable* if, and only if, all child segments of that given segment contained within the block are *flippable* – A *flippable* block defines a 2-separation where the boundary vertices are at the head vertex of the first segment in the block (with maximal $\prec$ precedence) and at the minimal $\prec$ precedence tail vertex of a segment within the block.

In case 2b (page 46), it is assumed that s_j has been assigned to a block defining which face (adjacent to s_i) it is embedded within and then, if s_k has an embedding, it must be assigned to be in the same block as s_j but within the opposing face.

Planar Embeddings of Segments

Considering case 2c (figures 4.2.7c-e) – where Class 4 segment s_j is embedded within a given face adjacent to s_i , the parent of Class 4 segment s_k , such that $s_i <^s s_j <^s s_k$ and that $v_t^j \preceq v_t^k \prec v_h^k \preceq v_h^j$; given this then s_j sub-divides the containing face adjacent to s_i and the head and tail vertices of s_k are both contained within the inside face.

Either: s_k 's head and tail vertices are both contained on the boundary s_j forms between its inside and outside adjacent faces – so s_k can be embedded within either region adjacent to s_j (case 2(c)ii, figure 4.2.7d); or at least one of s_k 's head or tail vertices is on the boundary of the inside face adjacent to s_j but is not contained on the boundary s_j forms between its inside and outside adjacent faces (cases 2(c)i & 2(c)iii, figures 4.2.7c & 4.2.7e).

Considering this latter sub-case: in most instances, s_k must be contained within the inside face relative to s_j ; however, if s_j is a descendant of a sibling $s_{j'}$ of s_k where $v_t^{j'} = v_t^j = v_t^k \prec v_h^{j'} = v_h^j = v_h^k$ (figure 4.2.8) then:

- All children of $s_{j'}$ are *flippable* – so s_j can be embedded within either the inside or outside face relative to $s_{j'}$;
- s_k 's head and tail vertices are identical to $s_{j'}$'s head and tail vertices so s_k can be embedded within either the inside or outside face relative to $s_{j'}$ (black and red embeddings); and
- If s_j and s_k are contained within the same face relative to $s_{j'}$ then s_k is within the inside face relative to s_j (highlighted in yellow).

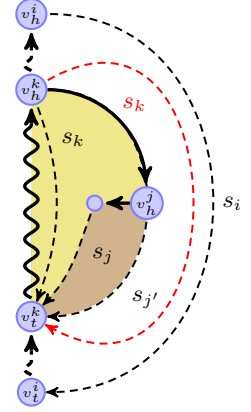


Figure 4.2.8: Permutable Segments s_k and s_j

This leads to the definition:

DEFINITION 4.2.39. Two Class 4 sibling segments s_j and s_k are defined to have a *permutable* embedding order (or, simply, to be *permutable*) when, without loss of generality, $s_j <^s s_k$ and where s_j and s_k have identical head vertices and identical tail vertices and have no descendants connecting to an ancestor other than at these shared head and tail vertices (or, succinctly, such that $v_t^j = v_t^k \prec v_h^j = v_h^k = v_h^j = v_h^k$).

COROLLARY 4.2.40. If segments s_j and s_k are *permutable* then $v_h^j = v_h^k$ so e_h^j and e_h^k are outbound from the same vertex and, by the composition of Class 4 segments, $v_t^j = v_t^k$ and $v_t^k = v_t^j$ therefore $v_{l_1}^j = v_{l_1}^k$ and $v_{l_2}^j = v_{l_2}^k$ therefore $e_h^j \parallel^* e_h^k$.

Considering two *permutable* sibling segments s_j and s_k such that $s_j <^s s_k$ then they have that ordering, given corollary 4.2.40, due to an arbitrary ordering assigned to the $\parallel^*$ edges e_h^j and e_h^k . Varying the arbitrary $\parallel^*$ order used to convert the partial $\prec^*$ order to a total $<^{**}$ order will, correspondingly, vary the $<^s$ order and s_k can be embedded before s_j . Thus if the segment ordered foremost is embedded and then the later ordered segment is embedded within the inside region relative to that foremost ordered segment then permuting the order of those segments generates all permutations of s_j being inside or outside of s_k while maintaining the process that one segment is never embedded outside of a previously embedded sibling.

CONCLUSION 4.2.41. A pair of segments, s_j and s_k where $s_j <^s s_k$ and $v_t^j \preceq v_t^k \prec v_h^k \preceq v_h^j$ (case 2c, page 46) is such that segment s_k always has an embedding within the inside face of s_j (and if s_j is a descendant of a sibling $s_{j'}$ of s_k then s_k is also embeddable within the inside face of $s_{j'}$).

A permutable segment can be embedded within both the inside and outside face relative to a sibling segment; an equivalent result can be achieved by always embedding the higher $<^s$ precedence permutable segment within the inside face relative to the lower $<^s$ precedence permutable segment and instead varying the order in which those permutable segments appear within the $<^s$ ordering – thus achieving a consistent process that segments are always embedded on the inside face of a lower $<^s$ precedence segment matching case 2c.

When a Segment and its Descendants are all Embedded

Considering a segment's *flippable* descendants (which includes all segments embedded within the outside region relative to that segment):

LEMMA 4.2.42. If all descendants of a parent segment s_p are embedded then each *flippable* child (and, recursively, descendant) segment s_c is either *fully embedded* or has $v_{l_1}^p = v_{l_1}^c$.

PROOF 4.2.43. By the definition of being *flippable*, each *flippable* child segment must either have:

- $v_h^p \preceq v_{l_1}^c$; or
- $(v_{l_1}^p = v_{l_1}^c) \wedge (v_h^p \preceq v_{l_2}^c)$.

The next segment s_n to be embedded must be a sibling of, or sibling of an ancestor of, s_c where $v_h^n \preceq v_h^p$ and so, respectively, then:

- $v_h^n \preceq v_h^p \preceq v_{l_1}^c$ – in which case, s_c is *fully embedded*; or
- $v_h^n \preceq v_h^p \preceq v_{l_2}^c$ and $v_{l_1}^p = v_{l_1}^c$. □

COROLLARY 4.2.44. This latter case can be considered further such that either:

- $v_{l_1}^p = v_{l_1}^c \preceq v_{l_1}^n \preceq v_h^n \preceq v_h^p \prec v_h^c$, in which case neither s_c nor s_p *conflicts* with s_n ;
- $v_{l_1}^n \prec v_{l_1}^p = v_{l_1}^c \preceq v_h^n \preceq v_h^p \prec v_h^c$, in which case both s_c and s_p *conflicts* with s_n ; or
- $v_{l_1}^n \preceq v_h^n \preceq v_{l_1}^p = v_{l_1}^c \preceq v_h^p \prec v_h^c$, in which case both s_c and s_p will be *fully embedded* when s_n is embedded.

Therefore, in that latter case, s_p and s_c will have an identical impact on the embedding of s_n .

This can be extended to consider, in exactly the same manner, when s_n is any sibling of, or sibling of an ancestor of, s_p ⁶ to show that in this general case that both s_p and s_c always have an identical impact on the embedding of s_n .

Given this then when s_p 's descendants are all embedded then all *flippable* descendants will be either *fully embedded* or otherwise inconsequential to the embedding of subsequent segments since s_p can be used to determine the impact upon the embeddings of those subsequent segments.

⁶ Descendants of those siblings are not considered since they will have higher $<^s$ precedence than s_n and so: any conflict between s_p , s_c and s_n will occur prior to embedding that descendant; and if there is no conflict then either s_p and s_c will be *fully embedded* or that descendant will always be attached to the cycle defined by s_n , or another descendant of s_n , which will always be inside of s_p and s_c .

DEFINITION 4.2.45. The definition of a *fully embedded* segment can be extended such that when a segment and its descendants have all been embedded then the descendants embedded within that segment's outside face can be considered to be *fully embedded* since they are inconsequential to subsequent embeddings.

Embedding a Class 4 Segment onto its Parent with Multiple Siblings

Previously, the embedding of a segment s_k within a region $\mathcal{R}$ relative to its parent s_p was considered compared to the segment with immediately preceding $<^s$ precedence; expanding this, then successively lower $<^s$ segments embedded within $\mathcal{R}$ can be considered until s_p is reached and, hence, if $\mathcal{R}$ is the inside of s_p , or if $\mathcal{R}$ is the outside of s_p and s_k is *flippable*, then s_k would always have an embedding.

Given that s_k is being embedded within a region $\mathcal{R}$ adjacent to its parent then the sequence of lower precedence segments already embedded within $\mathcal{R}$ is such that the preceding segments:

- Are siblings of, and siblings' descendants of, s_k where for s_k to be being embedded then, given the $<^s$ ordering, each preceding sibling segment's descendants must have been embedded; therefore, by lemma 4.2.42, each sibling's descendants embedded outside that sibling will be *fully embedded* (see definitions 4.2.30 & 4.2.45);
- Does not contain any segments which *conflict* (since each of those segments are embedded within the same region $\mathcal{R}$).

Therefore, each preceding segment within $\mathcal{R}$ is either *fully embedded* (either case 2a or, by definition 4.2.45, is a sibling's descendant embedded on the outside of that sibling) or is *partially embedded* (case 2c – either as a sibling of s_k or a sibling's descendant which is embedded inside that sibling).

Fully embedded segments have no impact on the embedding of subsequent segments thus to determine whether s_k has an embedding then only the *partially embedded* segments need be considered and given that each *partially embedded* segment is:

- Within $\mathcal{R}$ if it is the lowest $<^s$ precedence *partially embedded* segment within that sequence; or
- Inside the immediately $<^s$ preceding *partially embedded* segment (conclusion 4.2.41).

Therefore:

THEOREM 4.2.46. Each *partially embedded* Class 4 segment s_j embedded within a given region $\mathcal{R}$ adjacent to the embedding of its parent s_p is contained within a sub-division of $\mathcal{R}$ that is the inside region adjacent to any other *partially embedded* Class 4 segment s_i that is a descendant of s_p where $s_p <^s s_i <^s s_j$.

PROOF 4.2.47. Contained in the paragraphs above. □

THEOREM 4.2.48. Given a Class 4 segment s_k being embedded within a region $\mathcal{R}$ adjacent to s_k 's parent⁷, if s_k does not *conflict* with the maximal $<^s$ precedence *partially embedded* Class 4 segment s_j that is already embedded within $\mathcal{R}$ then s_k does not conflict with any other segment embedded within $\mathcal{R}$.

PROOF 4.2.49. Considering the classes of segments then:

- Each Class 1, 2 or 3 segment always has an embedding (as it is either *static* or *rotatable*);
- Each Class 4 segment always has an embedding within the inside region adjacent to its parent and, if *flippable*, also has an embedding within the outside region adjacent to its parent; and
- Once a Class 4 segment becomes *fully embedded* it cannot impact the subsequent embedding of a segment.

So, only the case where a Class 4 segment s_k is being embedded and there are siblings, or siblings' descendants, which are *partially embedded* needs considering as all other segments do not impact s_k 's embedding.

Given a sequence $\langle s_1, s_2, \dots, s_i, \dots, s_j, \dots \rangle$ of *partially embedded* Class 4 segments within a region $\mathcal{R}$ adjacent to a common ancestor (the parent segment of s_1) where $s_1 <^s s_2 <^s \dots <^s s_i <^s \dots <^s s_j$ then s_j is embedded inside of s_i is embedded inside of $\dots$ s_2 is embedded inside of s_1 . Given the formulation of case 2c, then the ascending precedence of the tail vertices of those segments corresponds to the ascending $<^s$ precedence of those segments:

$$v_t^1 \preceq v_t^2 \preceq \dots \preceq v_t^i \preceq \dots \preceq v_t^j \prec v_h^j$$

So, if a segment s_k (where $s_j <^s s_k$ and s_j is *partially embedded*) has an embedding that does not conflict with s_j then case 2c (page 46) also defines the relationship between s_j and s_k so $v_t^j \preceq v_t^k \prec v_h^k \preceq v_h^j$, hence, s_k does not conflict with segments $s_1 \dots s_i$.

Therefore, if when s_k is being embedded within a region $\mathcal{R}$ adjacent to its parent and there exists *partially embedded* Class 4 sibling, or sibling's descendant, segments with lower $<^s$ precedence than s_k which are embedded within $\mathcal{R}$ then, of those, the segment with maximal $<^s$ precedence determines whether s_k has an embedding relative to all other segments within $\mathcal{R}$. $\square$

COROLLARY 4.2.50. Given a sequence $\langle s_1, s_2, \dots, s_i, \dots, s_j, \dots \rangle$ of *partially embedded* Class 4 segments within a region $\mathcal{R}$ adjacent to a common ancestor (the parent segment of s_1) where $s_1 <^s s_2 <^s \dots <^s s_i <^s \dots <^s s_j$ then if segment s_k is tested for embedding and some of that sequence of segments are found to be *fully embedded* relative to s_k ; since the ascending precedence of the tail vertices of those segments corresponds to the ascending $<^s$ precedence of those segments then the segment with the maximal precedence tail vertex is the segment in the sequence with maximal $<^s$ precedence (i.e. s_j).

⁷ Assuming that if $\mathcal{R}$ is the outside region adjacent to s_k 's parent that s_k is *flippable* and can be embedded within $\mathcal{R}$.

Performing an Embedding (and Considering Blocks)

When embedding a Class 4 segment s_k into a region adjacent to its parent then segment s_{k-1} must just have been embedded and s_{k-1} is either:

- s_k 's parent – in which case, s_k always has an embedding within the inside region adjacent to its parent so can be trivially embedded forming a new block containing solely s_k (on the inside of the block since each segment always has an embedding within the inside region adjacent to its parent – theorem 4.2.24); or
- A descendant of s_k 's parent – in which case, there exists at least one *partially embedded* segment (either s_{k-1} or a sibling of s_k that is the ancestor of s_{k-1}) and, hence, at least one block of segments so s_k 's embedding can be considered in relation to those blocks.

This latter case can be considered further, however, a few useful properties of blocks can be noted first:

PROPERTY 4.2.51 (PROPERTIES OF BLOCKS). A new block is formed when a segment is embedded which does not conflict with any other previously embedded segment; given theorem 4.2.48 then if a segment does not *conflict* with the maximal $<^s$ precedence *partially embedded* segments on the inside and outside of a previously formed block then that segment must have an embedding on the inside face defined by the embedding of one of those segments. Hence, if there are multiple blocks then the segments of the most recent block are embedded within the inside face of segments of the preceding block and recursively considering successive preceding blocks then, recursively, the segments of each successor block are within the inside face of segments of the preceding block.

Using this and the fact that blocks only contain *partially embedded* segments then if a segment s_k is such that it does not conflict with the maximal $<^s$ precedence segment on one side of a block then, by theorem 4.2.48, it does not conflict with any other segment on that side of the block and, given that the minimal $<^s$ precedence segment on that side of the block does not conflict with any segments of any previous block then s_k does not conflict with segments in either side of previous blocks.

Returning to the embedding of a segment s_k in the case that at least one block exists relative to s_k 's parent then the first consideration is whether the segments which were *partially embedded* relative to the previous embedding of s_{k-1} are still *partially embedded* relative to s_k or if they are now *fully embedded*. This can be achieved by testing the v_h^k against the tail vertex of the maximal $<^s$ precedence segment on the most recently formed block's inside or outside and it will be found to be either:

- *fully embedded* with respect to s_k and so can be ignored in all future embeddings and removed from that block (if all segments of the block become *fully embedded* then no future embeddings can be impacted by the segments in that block so the block can be discarded and ignored) and the process can recursively consider the preceding $<^s$ precedence segment in the same region; or
- *partially embedded* with respect to s_k and so all lower $<^s$ segments within the same region that were *partially embedded* with respect to s_{k-1} will still be *partially embedded* with respect to s_k .

If all segments in a block become *fully embedded* then those segments will have no impact on the embedding of future segments and no additional segments will be added to the block. In this case, the block has performed its function for the purposes of testing for planarity and plays no further role in the test and the previous block can be, recursively, considered; however, the relationships between segments captured in that *fully embedded* block are useful in constructing a cyclic edge order to represent the embedding so while it can be ignored during the rest of the planarity test it should not be discarded.

If all segments of all blocks relative to s_k 's parent are *fully embedded* then the s_k can be embedded as if it was the first segment to be embedded onto its parent by creating a new block containing solely s_k on its inside.

If not all segment of all blocks relative to s_k 's parent are *fully embedded* then s_k can be considered against the most recently formed block which still contains *partially embedded* segments. Considering s_k against the maximal $<^s$ precedence *partially embedded* segments within the inside and outside of that block then either:

- s_k does not *conflict* with either side – in which case, no *partially embedded* segment defines which region adjacent to s_k 's parent s_k should be embedded in so the embedding of s_k will form a new block containing s_k on its inside (since s_k always has an embedding within the inside region relative to s_k 's parent).
- s_k conflicts with both sides – in which case, s_k does not have a planar embedding; or
- s_k conflicts with one side and not the other.

The first two cases are trivial in that s_k either has or does not have an embedding; the last case can be split into several sub-cases, where:

1. The conflict is on the outside and the inside contains a *partially embedded* segment which does not conflict;
2. The conflict is on the outside and the inside contains no *partially embedded* segments (so there is no conflict on the inside);
3. The conflict is on the inside and the outside contains a *partially embedded* segment which does not conflict; or
4. The conflict is on the inside and the outside contains no *partially embedded* segments (so there is no conflict on the outside).

In cases 1 and 3 then the block contains a *partially embedded* segment s_x which does not *conflict* with s_k . Given theorem 4.2.48 then if s_k was embedded within that face it would be embedded within the inside region relative to s_x and, recursively, within the inside region relative to each segment of lower $<^s$ precedence on the same side of that block. Since the lowest $<^s$ precedence segment on each side of the block must be within the inside region of both sides of any preceding block (otherwise the contents of those blocks would either be *fully embedded* or would *conflict*) then s_k must also be within the inside region of all segments (on either side) of all preceding blocks. Therefore, in cases 1 and 3, no additional blocks need be considered.

In cases 2 and 4 then the block contains no segments on the side of the block where there is no *conflict*. In this case, s_k is not necessarily within the inside face of the minimal $<^s$ precedence *partially embedded* segment within the block and, hence, s_k may also *conflict* with the segments of a preceding block. One could, iteratively, consider successively lower $<^s$ precedence segments within that block to determine whether s_k conflicts with them all or whether there exists a segment where s_k can be contained within its inside face; however, this process would require time linear to the number of segments in the block so is inefficient. A more efficient implementation is to assume that s_k conflicts with all segments in that block and immediately consider the segments in the preceding block. This process can be applied, iteratively, considering s_k against successive previous blocks:

- If s_k does not *conflict* with either side of the previous block then s_k can be embedded relative to that block (and all previous blocks) but no segment of that previous block defines which side s_k should s_k cannot be contained in that previous block and only within the most recent block – so no blocks need be considered other than the most recent block;
- If s_k *conflicts* with segments on both sides of the previous block then s_k does not have a planar embedding; and
- If s_k *conflicts* on a single side of the previous block then it must also *conflict* with all

segments in the most recent block⁸ and an attempt can be made to coalesce the two blocks:

- If the *conflict* within the most recent and previous blocks is on the same side then the blocks can be trivially coalesced.
- If the *conflict* is on opposing sides of the two blocks then if the most recent block is *flippable*⁹ then it can be *flipped* and coalesced with the previous block, however, if neither block is *flippable* then s_k is non-planar.

If the blocks can be coalesced then the coalesced block (now the most recent block) can be considered against cases 1-4 (page 56) to determine if further preceding blocks need to be considered and potentially coalesced (alternatively halting if all blocks relative to s_k 's parent have been considered).

Once the blocks have been considered and, if necessary, coalesced then either a determination will have been made that s_k does not have a planar embedding or s_k will be being considered for embedding within a single block that matches one of the cases 1-4 (page 56):

- In cases 1 and 2 (where s_k conflicts with a segment on the outside of the block but not the inside), given that s_k always has an embedding within the inside region relative to its parent, then s_k can be embedded on the inside of the block;
- In cases 3 and 4 (where s_k conflicts with a segment on the inside of the block but not the outside) then:
 - If s_k is *flippable* then it can be embedded within the outside of the block;
 - If s_k is not *flippable* but the block is then the block can be *flipped* and s_k embedded within the (now) inside of the block; otherwise
 - Neither s_k nor the block is *flippable* so s_k does not have a planar embedding.

One final point should be noted: When a segment s_k and its descendants are all embedded then there may exist segments which, at that time, are still *partially embedded* and may affect the later embedding of s_k 's siblings. By corollary 4.2.50 then any descendants of s_k embedded within its outside region will be (or are otherwise inconsequential and can be classed as) *fully embedded* and so only those segments within the inside region can be *partially embedded*. Since any descendant of s_k must be contained within the same region relative to s_k 's parent that contains s_k then those *partially embedded* segments inside of s_k can be moved from their containing block (relative to s_k) to the block containing s_k (relative to s_k 's parent) and on the same side of the block as contains s_k . Note: Moving these descendant segments from a block relative to s_k to a block relative to s_k 's parent does not change whether either of those blocks is *flippable*.

By iterative application of this, considering segments in $<^s$ precedence order, then all segments will be considered and either embedded or found not to have a planar embedding.

⁸ Since v_t^k precedes the tail vertex of the maximal $<^s$ segment of that previous block and the segments of the most recent block are within the inside region of the segments of that previous block so their tail vertices must have higher precedence and also conflict.

⁹ If the preceding block is *flippable* then, since the segments of the most recent block are within the inside face of segments of that previous block then the most recent block must also be *flippable*.

Example Embedding

Example 4.1.31 (page 38) (repeated in figure 4.2.9) gives the segments generated for the sample graph shown in figure 3.1.1 (page 28). Figure 4.2.10 shows the embedding of these segments and figure 4.2.11 shows the corresponding contents of the blocks; demonstrating a lot of the different aspects of embeddings previously described.

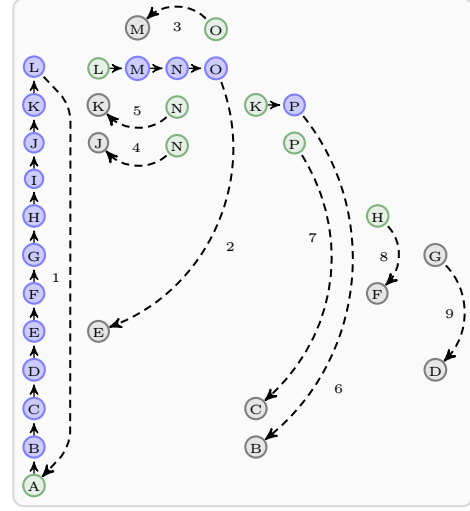


Figure 4.2.9: Segments of Sample Graph
Numbered in $<^s$ Order

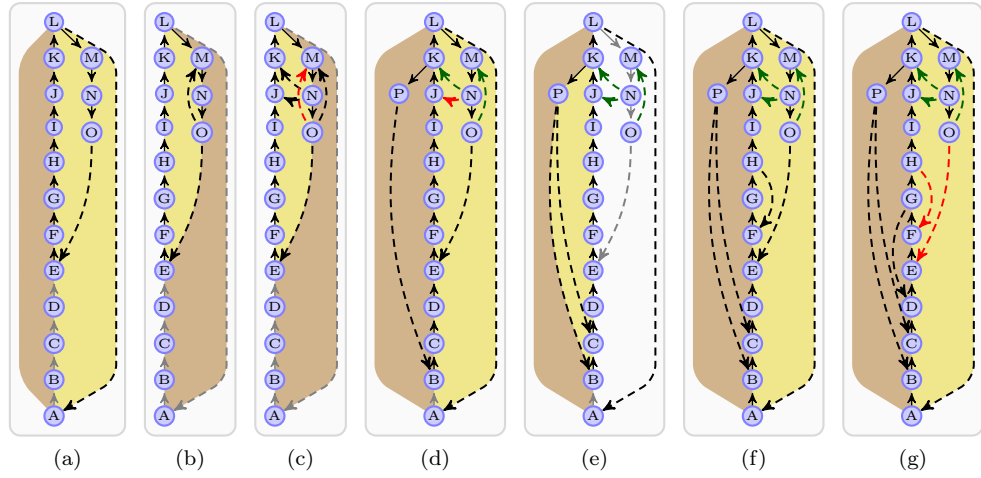


Figure 4.2.10: Embedding of Segments of Sample Graph

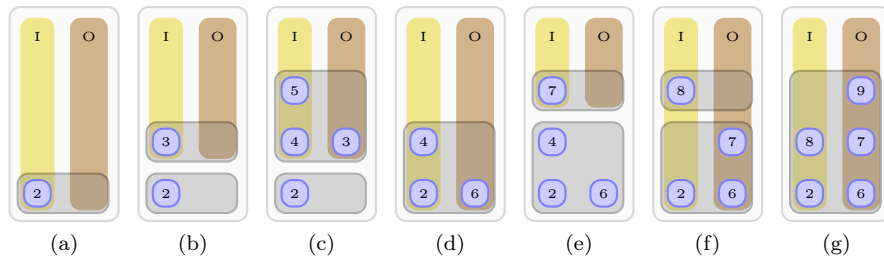


Figure 4.2.11: Contents of Blocks during Embedding From Figure 4.2.10

Examining the embedding in more depth:

- (a) shows the embedding of the root segment (having no parent so defining no block) which defines an inside (tan) and outside (brown) region. The second segment is then embedded forming a new block (relative to the root segment) which contains that segment on the inside – ordinarily this segment would be *flippable* but this is superseded by a requirement for it to be *static* to define which region adjacent to its parent is the inside.
- The cycle formed by this second path defines two faces (tan and brown shown in (b)) and within that the third segment can be embedded. There are no previous blocks defined relative to its parent (the previous block being relative to the root segment) so a new block is formed with the third segment on its inside – both the third segment and its containing block are *flippable*.
- (c) shows the embedding of the fourth and fifth segments. When the fourth segment is embedded it conflicts with the third (red) and must be added to the opposite side of the block from that segment; however, the fourth segment is not *flippable* so, instead, the block containing the third segment is *flipped* and the fourth segment added to the (now) inside (in the process making that containing block non-*flippable*). The fifth segment conflicts with the third (on the outside) but not the fourth (on the inside) so can be embedded on the inside of that block.
- Considering (d): the algorithm has processed the second segment and its descendants so the segment (third) on the outside is now *fully embedded* (green) and the segments on the inside (fourth and fifth) are moved from their containing block to the block containing their ancestor (the second segment) on the same side as that ancestor.
Processing the sixth segment: the first consideration is whether any previous segments have changed status and, relative to that segment, the fifth segment is now *fully embedded* while the fourth is still *partially embedded*; after that then the sixth segment can be embedded and it conflicts with the fourth segment (inside) and there are no segments on the outside of the block (nor any preceding blocks) – since the sixth segment is *flippable* then it is embedded on the outside of the preceding block.
- The cycle formed by the sixth path defines two regions (tan and brown shown in (e)) and within that the seventh segment can be embedded. There are no blocks relative to the sixth stack so a new block is formed with that seventh segment on its inside.
- In (f) the sixth segment and its descendants have been processed so the descendants on the inside of the sixth segment (i.e. the seventh segment) are moved from that block to the block containing the sixth segment and on the same side as that segment (i.e. the outside).
- The eighth segment is then considered: the fourth segment becomes *fully embedded*, removing it from the block, and then its embedding is tested and found not to *conflict* with either of the maximal $<^s$ segments on the inside or outside (being able to be contained within the inside face relative to both the second and seventh segments) so a new block is formed containing the eighth segment on its inside.
- The final (ninth) segment's embedding (g) is tested against the most recent block and found to *conflict* with the eighth segment on the inside. However, the outside of the block contains no segments so it is also tested against the previous block where it finds a *conflict* with the second segment (inside) but no *conflict* with the seventh segment (outside). Given this, then the two blocks are coalesced – since both *conflicts* occur on the inside then no block is *flipped* and the ninth is embedded on the outside of that block (since it is *flippable*).
- The algorithm then terminates, having embedded all segments, finding the graph planar.

4.3 Permutations of Embeddings

During the embedding several statuses can be assigned to segments or blocks:

Static The root segment and, if the root segment is not of Class 2, its child with minimal $<^s$ precedence both have a *static* embedding – such that the root segment has a fixed cyclic direction around its inside face and that the child (or, for a Class 2 root segment, the leading path of bridge edges) is contained in this inside face (but does not necessarily have a fixed cyclic direction).

Rotatable Where any non-*static* Class 1, 2 or 3 segment can be embedded within any face adjacent to its head vertex.

Reversible Where any non-root Class 2 or 3 segment can have the direction of their edges around their inside face *reversed*.

Flippable A block of Class 4 segments (relative to a given segment) is *flippable* if all child segments (of that given segment) added to that block can be embedded within both the inside and outside regions adjacent to that given segment (i.e. the segments of the block and the path within that given segment connecting the minimal and maximal precedence vertices incident to those segments can be transformed via a *Whitney flip* to produce an alternate embedding).

Permutable A pair of Class 4 segments s_x and s_y are permutable if they are siblings and if s_x has both an embedding within the inside region and an embedding within the outside region relative to s_y , or vice versa (equivalently, either: $v_h^x = v_{l_2}^x = v_h^y = v_{l_2}^y$ and $v_t^x = v_t^y$; or, $v_h^x = v_{l_2}^x = v_h^y = v_{l_2}^y$ and $e_h^x \parallel^* e_h^y$).

Outer Reversible

An additional case (linked to the *reversible* status) where the embedding can be permuted¹⁰ is for a Class 4 segment that is the descendant of, and has identical tail vertex as, a non-root Class 2 or 3 segment. In this case, that descendant segment has three (instead of the normal maximum of two for a non-*permutable* Class 4 segment) possible embeddings: inside, outside clockwise and outside anticlockwise relative to the inside of the cycle defined by that Class 2 or 3 segment¹¹.

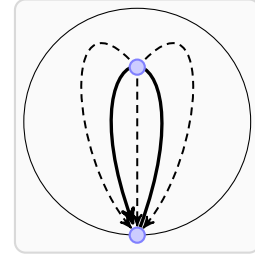


Figure 4.3.1: Possible Embeddings of a Class 4 Child Segment Attached to a Non-Root Class 3 Segment Where Both Segment Have A Common Tail Vertex

Given the construction of the segments then the minimal precedence vertex that a descendant of a Class 2/3 segment (where that descendant is outbound from a vertex

¹⁰ All Class 4 children of a Class 2 or 3 segment are *flippable* unless they are *static*. A non-root Class 2 or 3 segment, of which an *outer reversible* segment is a descendant of, could be a *static* child of the root segment but in no case can any of its descendants be *static* thus no *outer reversible* segment is static nor has a *static* sibling to conflict with. Therefore all Class 4 children of a non-root Class 2 or 3 segment are *flippable* and so if their descendants are planar then as the children are *flipped* the descendants will swap regions too and have an embedding within both the inside and outside regions relative to that Class 2 or 3 segment.

¹¹ It should be noted that the cyclic direction of the segment needs to be relative to the inside region of the Class 2 or 3 cycle as the direction of the Class 4 segment around the inside region it defines is (when embedded outside its parent) constant regardless of whether it is clockwise or anticlockwise relative to the cycle's inside region.

on the cycle defined by that Class 2/3 segment) could potentially reach is the tail of that Class 2/3 segment. Therefore, if a descendant segment's tail vertex is the tail vertex of a Class 2/3 ancestor then all ancestors of that descendant which are also descendants of that Class 2/3 segment will also have the same tail vertex and be *outer reversible*.

Figure 4.3.1 gives a graphical example of this case showing three dashed lines representing the possible embeddings of a Class 4 segment outbound from a Class 3 segment and sharing a common tail vertex.

Thus, once all segments are embedded, if a segment is *rotatable*, *reversible* or *permutable* or a block of segments is *flippable* then alternate embeddings can be generated of the graph.

4.3.1 Handling Flippable Segments

If a block is *flipped* then the segments embedded on the inside are swapped to be on the outside and vice versa. Given that *flipping* a block is a binary state then if there are f *flippable* blocks there are 2^f permutations of the embedding due to *flipping* blocks.

Whether f *flippable* blocks are *flipped* or not can be represented by an f -bit binary number and generating all permutations of *flips* can be achieved by incrementing that binary number and querying the status of each bit.

4.3.2 Handling Permutable Segments

If a group of segments are all *permutable* then the head edges of those segments all have incomparable ($\parallel^*$) TT-precedence and the initial $<^s$ precedence of those segments is dependant on an arbitrary order assigned to those head edges. Since a *permutable* segment (and its descendants) does not *conflict* with any other segment it is *permutable* with (nor that segment's descendants) and that all segments within the same *permutable* group share the same head and tail vertices (and their descendants are not connected to any ancestors at any other points) then if one order of the *permutable* segments produces a planar embedding then any order of those *permutable* segments will produce a planar embedding. Given this, then it is unnecessary to retest for planarity when embedding *permutable* segments in an alternate ordering – and by extension if, when testing for planarity, one segment of a *permutable* group has a planar embedding relative to the lower $<^s$ segments then all segments of that *permutable* group do (however, this does not prevent the descendants of any single *permutable* segment from forming a non-planar sub-graph).

Considering the degenerate case from page 44 (figure included right), then the permutation of the embedding of two segments (black and red/green – highlighting two possible embeddings of that second segment on either side of the first) between vertices u and v can be considered such that segments are always embedded within a face (yellow) containing an edge inbound to and an edge outbound from the embedding segment's head vertex (u).

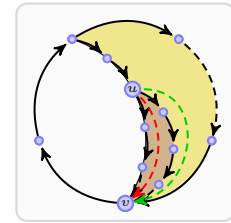


Figure 4.3.2: Degenerate Embedding Case

- The green permutation of the embedding can be achieved by default by embedding the black and then embedding the second within the yellow face.
- The red embedding requires embedding within a face where both edges on the boundary of the face are outbound from u ; however, by permuting the order of the segments such that the red embedding is performed before the black then both embeddings are within faces with appropriate boundary directionality.

In this example, additional embeddings are possible if one or both of those segments are considered flipped (within the white face) and this results in a special case (considered on the following page).

Given a group of p *permutable* segments then there are $p!$ ways to generate an order of those segments. An order of a group of *permutable* segments can be represented by a sequence and by swapping adjacent elements of the sequence then all $p!$ permutations can be generated in $p!$ swaps using the Steinhaus-Johnson-Trotter algorithm [38, 76] – this implementation is discussed in more depth in Chapter 5 (on page 90). In this case, the cost to generate a different permutation of the embedding is the cost to swap two adjacent elements in the list or, more specifically, the cost of finding the element within the list that is to be swapped to obtain the next permutation and this is shown to be a constant average time.

4.3.3 Handling the Special Case (Permutable and Flippable and ...)

A special case occurs when the group of *permutable* segments are also *flippable* and do not *conflict* with any other segments. In this case, all those *flippable* segments can be embedded within either the inside or outside region relative to its parent (since they are not restricted by either their parent or a conflict). Thus the group of *permutable* segments can be partitioned into two disjoint sub-groups corresponding to the sides of the parent within which they are embedded; each of those sub-groups can be re-ordered independently and segments can be moved between sub-groups.

A simple method of dealing with this special case is to:

1. Exclude the block containing that group of *permutable* segments from the n -bit binary number representing the status of the *flippable* blocks; and
2. Instead, represent those segments with a sequence containing those segments and that group's parent segment. If a segment precedes (succeeds) its parent in the sequence then it is, arbitrarily, in the outside (inside) region relative to its parent and the order of the segments within the sub-sequences preceding and succeeding the parent determines the embedding order within that region.

Thus, for a group of p *permutable segments* then, in this special case there are $(p + 1)!$ possible arrangements of embeddings.

When the Special Case Contains a Static Segment

A further sub-case of that special case occurs when the group of *permutable* segments contains a *static* segment¹². In this specific instance then the *permutable* segments can move freely between the inside and outside regions relative to the root segment with the exception of that *static* segment which must always be within the inside region; however, the order of those sub-groups of inside and outside segments can be freely re-ordered (including that *static* segment).

This can be represented as per the special case, above, with the added constraint that the parent (root) segment and the *static* child segment are initially ordered such that precedence within the sequence of those two segments defines the *static* child segment to be within the inside face and, during subsequent swaps of adjacent segments to reorder the sequence, that the parent and *static* child segments are never swapped. In this sub-case, for a group of p *permutable segments* then there are $1/2(p + 1)!$ possible arrangements of embeddings (since there are $(p + 1)!$ unconstrained

¹²In which case, the group's parent is the root segment and is of Class 3 – since the only Class 4 segments are defined to be permutable so the root segment must be of Class 2 or 3 to have a Class 4 child and if the root segment was of Class 2 then its child would not be *static*.

orderings and in half those orderings the *static* child will have an invalid precedence compared to its parent). This can be implemented through careful initialisation of the sequence and by using the Steinhaus-Johnson-Trotter permutation algorithm as discussed in the next chapter.

Figure 4.3.3 shows an example graph where the edges are coloured by segment such that: the root segment is black; it has two *permutable*, *flippable* children which do not *conflict* with any other segments that are coloured orange (*static*) and blue (not *static*); and each of these three segments has a pair of children which *conflict* but are both *flippable* thus defining three *flippable* blocks. There are 2^3 embeddings from varying whether blocks are *flipped* and $3!/2$ embeddings from re-ordering *permutable* segments giving a total of $2^3 \times 3!/2 = 24$ different embeddings (shown in figure 4.3.4).

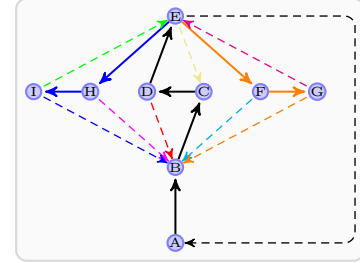


Figure 4.3.3: Example Graph with 3 *Flippable* Blocks and 2 *Permutable*, *Flippable*, *Non-Conflicting* Segments Including the *Static* Child Segment

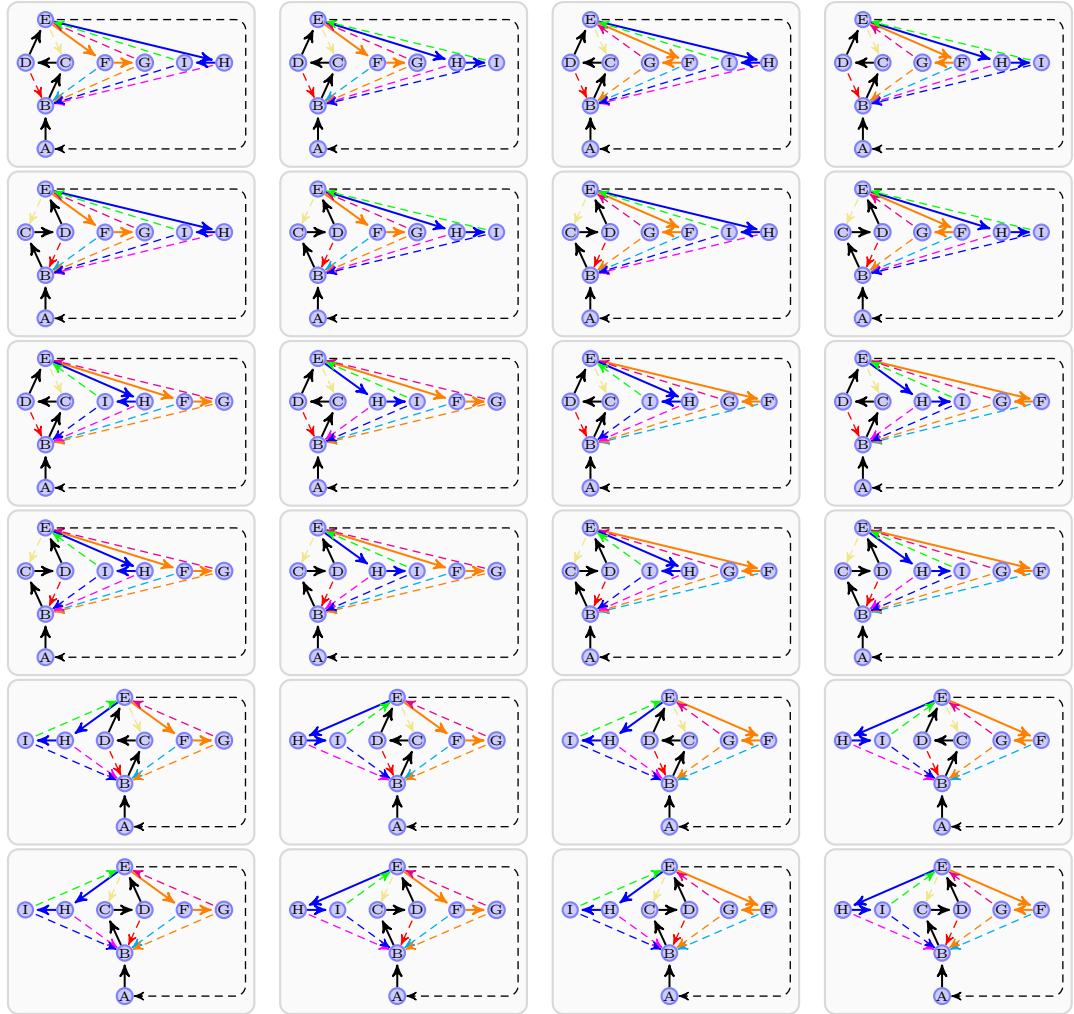


Figure 4.3.4: All Permutations of Example Graph in Figure 4.3.3

4.3.4 Handling Reversible Segments

Whether a segment's cyclic direction is *reversed* is, similar to whether a block is *flipped*, a binary state. Given this then if there are r *reversible* segments then there are 2^r permutations of the embedding due to *reversing* segments.

Whether r *reversible* segments are *reversed* or not can be represented by an r -bit binary number and generating all permutations of *reversals* can be achieved by incrementing that binary number and querying the status of each bit.

4.3.5 Handling Rotatable Segments

Initially, Class 1, 2 or 3 segments can be considered independently such that each segment either: forms the root segment of a biconnected component (with, Class 2, or without, Class 3, a leading path of bridge edges) with its Class 4 descendants; or forms a simple path (Class 1).

Rotatable segments can be partitioned into two sub-categories where the segment and its descendants connects to its head vertex with: a single (bridge) edge (Class 1 or 2 segments); or, multiple edges (Class 3 segments). Further to this, *rotatable* segments can be considered grouped by sub-category and by distinct head vertices.

When considering the embedding of a group of *rotatable* segments with a common head vertex v then the assumption is made that there is a fixed cyclic order of edges incident to v generated by an fixed embedding of the *rotatable* segment's common parent segment (containing v) and all that parent's Class 4 descendants which are not descendants of a *rotatable* descendant of that parent. This can be fixed cyclic order can be represented by the sequence $\langle e_1, e_2, \dots, e_n \rangle$ where e_1 is embedded such that it is immediately clockwise about v from e_n and where for all $2 \leq x \leq n$ then e_x is embedded such that it is immediately clockwise about v from $e_{(x-1)}$.

Onto this fixed cyclic edge order can be attached:

1. The group of Class 3 *rotatable* segments and their descendants; then
2. The group of Class 1 and 2 *rotatable* segments.

Handling Non-Root Class 3 Segments and Descendants

Considering the attachment of a non-root Class 3 segment to its parent segment (and the biconnected component formed by that parent and its Class 4 descendants). Initially, each non-root Class 3 segment and its Class 4 descendants can be considered independently of its parent assuming

A Class 3 segment contains two edges (its head and tail) incident to its head vertex; additionally, its descendants can connect to the head vertex either within the inside or outside region relative to that segment. Any descendant of that Class 3 segment whose tail vertex is that Class 3 segment's head vertex must, by the relative precedence of the descendant's head and tail vertices, be a Class 4 segment; therefore, those descendants must be, by definition *outer reversible* and may be *flippable* or *permutable* but cannot be *rotatable* or *reversible*.

Since an *outer reversible* segment has one embedding inside the cycle defined by the class 3 segment then it always defines permutations of the embedding and defines them in different numbers depending on whether it is inside or outside of the cycle. This is shown in figure 4.3.5 (below) since when the three dashed segments, connecting to the head vertex of the Class 3

cyclic segment, are in its outside face there are four (a-d) possible embeddings (one for the cycle and one for each of those segments in the outside face) but when they are flipped to be embedded within the inside face there is a single embedding (e).

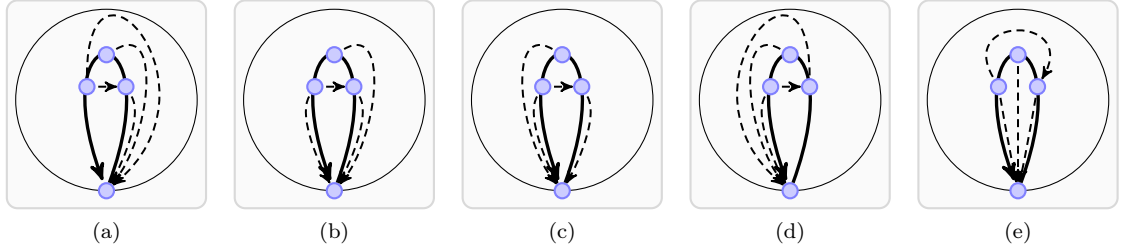


Figure 4.3.5: Four Possible Embeddings (a-d) Within the Outside Face and When *Flipped* (e) a Single Embedding Within the Inside Face

Considering the embedding of a sequence $\langle s_1, s_2, \dots, s_m \rangle$ of m Class 3 segments with a common head vertex where the sequences $\langle i_1, i_2, \dots, i_m \rangle$ and $\langle o_1, o_2, \dots, o_m \rangle$ represent the numbers of *outer reversible* segments emanating from each correspondingly numbered Class 3 segment within, respectively, the inside and outside faces of those segments. Also given that the head vertex has a cyclic order $\langle e_1, e_2, \dots, e_n \rangle$ of n edges already embedded where those edges represent the embedding of the parent segment and Class 4 descendants (as described above) then each Class 3 segment must be embedded between an edge and, without loss of generality, the edge immediately clockwise from it (since those edges initially considered form part of a biconnected component and so a Class 3 segment's head and tail edges cannot "straddle" one or more edges previously embedded as part of a biconnected component without crossing the boundary of a face and being non-planar).

When the first segment s_1 is embedded, it can be contained in any of n faces and, given its embedding within one such face in a fixed cyclic direction, then there will now be $n + 2$ faces. Considering the descendants of s_1 then if none terminating at its head vertex (i.e. $i_1 + o_1 = 0$) then nothing else needs to be considered; however, if it has descendants terminating at its head vertex then all of those descendants can be *flipped* so there are $(1 + o_1)$ possible embeddings when not *flipped* and $(1 + i_1)$ possible embeddings when *flipped*. Giving a total of $n(2 + i_1 + o_1)$ possible embeddings (or n possible embeddings if there are no descendants). This is further complicated with those descendants are not *flipped* as a group but there are multiple blocks which can be *flipped* independently.

When the second segment s_2 is considered, it can be embedded between any pair of cyclicly adjacent edges around that head vertex; which is now $(n + 2 + i_1 + o_1)$ edges. However, it can also be embedded such that one of the interior faces defined by its embedding contains the embedding of s_1 . Therefore, assuming $i_2 + o_2 > 0$ then, s_2 has $(n + 3 + i_1 + o_1)(2 + i_2 + o_2)$ possible embeddings.

This rapidly becomes a very large number of possible embeddings and the embedding of successive segments can depend on the embedding of previous segments and the arrangement of their descendants.

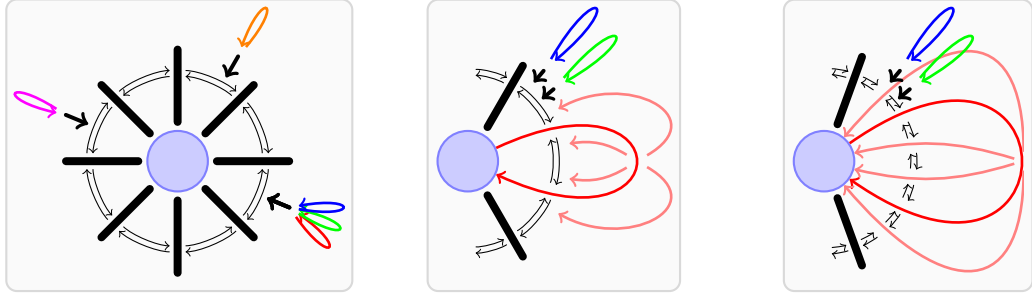


Figure 4.3.6: Example Permutations of Class 3 Segments

Figure 4.3.6 gives an example, spread over three panels, of this. A vertex is first considered with all its ancestors and incident class 4 children embedded (black edges) and then the class 3 children are attached. The example shows the attachment of five class 3 children (red, green, blue, orange and purple) in an arbitrary attachment. Two, orange and purple, are attached between distinct pairs of adjacent incident edges, so assuming a fixed cyclic direction, have a unique embedding; however, the other three are all embedded between the same pair of adjacent incident edges and can be attached in any order as well as having regard for descendants of those segments. The middle panel shows the attachment of the red class 3 segment (with a given cyclic direction) and some example (faded red) descendants (with a fixed outer reversible order of attachment). The right panel gives an example attachment where the remaining two class 3 segments are attached between different pairs of adjacent edges – however, those segments could have equally been embedded between any other pair of adjacent edges (even between the same pair, repeating the consideration given in the middle panel).

Therefore, the possible embeddings of a class 3 segment is dependant on prior embeddings and unlike class 1, 2 or 4 segments cannot be considered independently. For these reasons, there is not an immediate solution to the problem of predicting the total number of embeddings for multiple Class 3 segments (especially with *outer reversible* descendants) so it is difficult to prove that, on average, a time bound linear to the number of permutations and the number of edges (or segments) can be achieved.

One possible avenue of investigation for creating an algorithm to all possible embeddings of class 3 segments is to form the sequence $\langle e_1, e_2, \dots, e_n, s_1, s_2, \dots, s_m \rangle$ and then swapping adjacent elements of the sequence, using the Steinhaus-Johnson-Trotter permutation algorithm, with the added constraint that e_1 precedes e_2 which both precede e_3 which ... all precede e_n within that sequence (i.e. the algorithm never swaps e_{x-1} and e_x) but it is allowed to intersperse the order of edges with segments in any order. Given this then $(n+m)!/n!$ permutations of that sequence can be generated defining which segments are embedded between which pair of cyclicly adjacent edges and the relative order of those segments between each pair.

Considering the sub-sequence $\langle \dots, e_{x-1}, s_a, s_b, s_c, \dots, e_x, \dots \rangle$ then it can be considered to mean that s_a is embedded clockwise ($\curvearrowright$) of e_{x-1} and s_b is embedded clockwise or inside of s_a and s_c is embedded clockwise or inside of s_b . The state where s_b is considered anti-clockwise ($\curvearrowleft$) or outside of s_a can be achieved when the sequence is re-ordered such that s_b precedes s_a ; thus offering some simplification.

From this then those segments embedded within the same face (bounded by two adjacent incident edges) can be considered independently from all other segments embedded within different faces. The first segment s_a has, given a fixed cyclic direction, a unique embedding and its children can be embedded in a given outer reversible manner. Then the next segment s_b can be embedded, relative to the vertex being considered, clockwise of the most clockwise descendant of s_b (or s_b most clockwise edge if it has no descendants embedded on the outside) and achieving all

possible embeddings can be achieved by iterating, anti-clockwise, through each adjacent face formed by s_a and its descendants. Similarly s_c can be embedded clockwise and outside of the initial placement of its predecessor s_b and all possible embeddings can be achieved by iterating through all faces that are inside of, or clockwise of, its predecessor s_b .

The problem for this method is to create a data structure that will be able to handle the iterative repositioning of s_b , s_c and their descendants and this will require further work to prove that this process for embedding multiple class 3 segments can be achieved in linear time. As such, the empirical testing performed in chapter 7 does not consider multiple permutations of class 3 segments.

Handling Class 1 and 2 Rotatable Segments

For this, it can be assumed that all non-Class 1 or 2 segments have been embedded and there is a fixed cyclic order $\langle e_1, e_2, \dots, e_n \rangle$ for those edges.

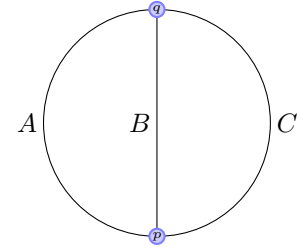
A Class 1 or 2 segments's head vertex is incident to a single edge within the sub-graph formed by that segment and its descendants. Considering two Class 1 or 2 segments s_1 and s_2 , outbound from a common head vertex, then when embedded: s_1 is in the outside region relative to s_2 ; and s_2 is in the outside region relative to s_1 – since a Class 1 or 2 segment cannot be inside a region defined by another Class 1 or 2 segment then the issues with Class 3 segments are not present and they have a comparably trivial embedding.

Given a sequence $\langle s_1, s_2, \dots, s_m \rangle$ of m Class 1 and 2 segments then the ordering of the embedding of these segments about their head vertex can be handled simply by constructing the $\langle e_1, e_2, \dots, e_n, s_1, s_2, \dots, s_m \rangle$ and then swapping adjacent elements of the sequence (for example, using the Steinhaus-Johnson-Trotter permutation algorithm) with the added constraint that e_1 precedes e_2 which both precede e_3 which ... all precede e_n within that sequence (i.e. the algorithm never swaps e_{x-1} and e_x) but it is allowed to intersperse the order of edges with segments. The result of this ordering is that $(n+m)!/n!$ permutations of that sequence can be generated defining which segments are embedded between which pair of cyclicly adjacent edges and the relative order of those segments between each pair.

4.4 Generating a Cyclic Edge Order

Given an embedding of a graph then a cyclic edge order of that embedding is an clockwise/anti-clockwise ordering of the edges incident to each vertex which represents that embedding. Algorithmically, it can be represented such that the edges incident to each vertex define a doubly-linked circular list such that each forward (backwards) link from a given edge in the list represents the link to the next edge clockwise (anti-clockwise) about that vertex; therefore each non-looping edge is contained in two distinct doubly-linked lists (one for each incident vertex) and each looping edge has two entries in the same doubly-linked list.

It is useful to recall a corollary [32, pg. 551-552] to the Jordan Curve Theorem that: Jordan arcs A , B and C , which have the same endpoints (p and q) but do not otherwise intersect, then if the clockwise orientation of the arcs about p is in the order $\langle A, B, C \rangle$ then the clockwise orientation of those arcs about q is in the order $\langle A, C, B \rangle$ (figure 4.4.1).



In addition, it is useful to define the concept of *cyclic direction* for cyclic/biconnected segments:

DEFINITION 4.4.1. Given a Class 2, 3 or 4 segment then that segment defines the boundary between an inside and an outside region (from definition 4.2.18) and that segment's *cyclic direction* is defined to be whether its edges are directed clockwise or anti-clockwise around that inside region.

4.4.1 Initial Cyclic Edge Order (Solely Within a Segment)

Not considering the attachment of descendant segments to their ancestors then:

DEFINITION 4.4.2. Each non-root vertex has an immediately preceding (tree) edge inbound to that vertex which is contained in the same segment that can be defined to be its *parent* edge for that vertex. Similarly, each degree 2 or more vertex has an immediately succeeding (minimum $<^{**}$ precedence) edge outbound from that vertex which is contained in the same segment that can be defined to be its *stem* edge. Together these edges define the initial cyclic edge order for a vertex such that the next edge both clockwise and anti-clockwise from the *parent* (*stem*) edge is the *stem* (*parent*) edge.

This initial cyclic edge order can be expanded as child segments and their descendants are considered for attachment to the embedding.

Considering the segment's tail when it is contained in the same segment then, in the case of:

- A Class 1 segment (assuming the graph is not a single vertex), its tail vertex has degree 1 so, trivially, the cycle of edges about that tail vertex consists of a single edge;
- A Class 2 segment, its tail vertex has both a *parent* and *stem* edge, so its tail is a third edge incident to this vertex; if the segment has a clockwise (anti-clockwise) cyclic direction then the tail edge can be inserted into the tail vertex's cyclic edge order clockwise (anti-clockwise) from the *stem* edge; and
- A Class 3 root segment then the root vertex has no immediately preceding edge but, within that segment, the tail (cotree) edge is inbound to the root vertex and, for simplicity, the *parent* edge of the root vertex can be defined to be the root segment's tail edge.

The attachment of head edges and other cases of tail edges is considered in more depth in the following pages.

4.4.2 Generating a Cyclic Edge Order for a Biconnected Component

A biconnected component is formed of a cycle (defined by a Class 2 or 3 segment) and zero or more Class 4 segments.

Calculating the Cyclic Direction of a Class 4 Segment

A Class 4 segment branches from its parent (which must be of Class 2, 3 or 4 and defines the boundary between an inside and an outside region within one of which the Class 4 segment is embedded) therefore considering the cyclic direction of both segments (figure 4.4.2) then:

- If the parent is clockwise (black dashed arc) and the child is inside/outside (red/blue dashed arc, respectively) then that child's cyclic direction is clockwise/anti-clockwise (respectively); and
- If the parent is anti-clockwise (grey dashed arc) and the child is inside/outside (blue/red dashed arc, respectively) then that child's cyclic direction is anti-clockwise/clockwise (respectively).

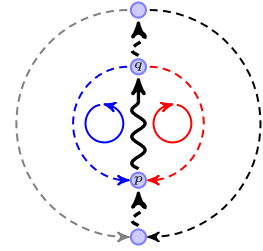


Figure 4.4.2: Segments With Clockwise (Red) and Anti-Clockwise (Blue) Cyclic Direction Connecting to Parent With Clockwise (Black) or Anti-Clockwise (Black and Grey) Cyclic Direction

Alternatively, this can be simply defined such that:

DEFINITION 4.4.3. Given a Class 4 segment that is embedded within the inside (outside) region defined by its parent then that segment has the same (opposite) cyclic direction as its parent.

Therefore, by assigning an arbitrary direction to the Class 2 or 3 segment that is the root segment of the graph (or of the biconnected component) and having performed an embedding of the graph (such that each Class 4 segment is assigned to be either on the inside or outside region of its parent) then each Class 4 segment will have a well defined *cyclic direction*.

Generating a Cyclic Edge Order for Class 4 Segments

Considering the $<^s$ order then the first (designated as “that parent”) segment to contain a vertex v defines its initial cyclic order (usually defining an inbound and an outbound edge).

A Class 4 segment s_x can either connect to a vertex v in that parent via:

- Their head edge where v is s_x 's head vertex; or
- Their tail edge where v is s_x 's tail vertex and can be sub-categorised into:
 - When s_x has no ancestor (except that parent) inbound to v ;
 - When s_x has a Class 4 ancestor (other than that parent) inbound to v ; or

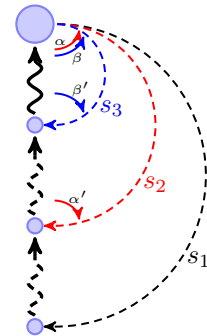


Figure 4.4.3: Angles From Parent & Stem Edges To Segments With Increasing $<^s$ Precedence

- When s_x has a Class 3 ancestor (other than that parent) inbound to v so s_x is *outer reversible*.

Considering the attachment of s_x at its head vertex then either s_x is the first child attached to that side of the segment at v or there will exist a previous segment and, by theorem 4.2.48, the latter segment will be added within the inside face of the previous. In both cases:

DEFINITION 4.4.4. A segment's head edge can be added to the cyclic edge order about its head vertex v by inserting it into the cyclic order adjacent to v 's parent edge in the opposite cyclic direction to the embedding of that segment.

Figure 4.4.3 gives a graphical representation of this such that the parent segment s_1 is embedded in a clockwise cyclic direction and segment s_2 is embedded within the inside region so has the same (clockwise) cyclic direction and the head edge is the opposite (anti-clockwise) cyclic direction from the head vertex's parent edge (angle α). Segment s_3 is also embedded in the same region; is outbound from the same head vertex; and has a the same clockwise cyclic direction. Given this then s_3 's head edge is also anti-clockwise (angle β) from the head vertex's parent edge (and it can be noted that $\beta < \alpha$).

Considering the attachment of s_x at its tail vertex v where s_x has no ancestor that inbound to v (except that parent) then either:

- s_x is a child of that parent and no ancestor exists (and can note that s_x is of Class 4 so $v \prec v_h^x$ and the most recent outbound edge from v is contained in that parent); or
- s_x is a descendant of a child s_c of that parent (and can note that $v = v_t^x \preceq v_h^c \prec v_h^x$ so either: $v = v_h^c$ and the most recent outbound edge from v is in s_c ; or $v \prec v_h^c$ and the most recent outbound edge from v is contained in that parent).

In both case, s_x can be considered relative to the most recent outbound edge e from v where that most recent edge is contained in a segment designated s_r then s_x must be either: a child of s_r ; or a descendant of a child of s_r and, in this latter case, since s_x terminates at v but no other ancestor of s_x does then that child of s_r must terminate at a vertex preceding v and so s_x is not *flippable* and is within the inside region of that child of s_r (and all other ancestors of s_x back to that child) so has the same cyclic direction as that child. Therefore:

DEFINITION 4.4.5. A segment's tail edge, where no ancestor of that segment contains an edge inbound to that segment's tail vertex (except within the first segment to contain that vertex), can be added to the cyclic edge order about its head vertex v by inserting it into the cyclic order adjacent to the most recent outbound edge from v in the same cyclic direction as the embedding of that segment.

Figure 4.4.3 gives a graphical representation of this such that the parent segment s_1 is embedded in a clockwise cyclic direction and segment s_2 is embedded within the inside region so has the same (clockwise) cyclic direction and the tail edge is in the same (clockwise) cyclic direction (angle α') from the head vertex's stem edge which is the most recent outbound edge.

Considering the case where s_x is being attached at its tail vertex v and s_x has a Class 4 ancestor s_a (excluding that parent segment) that is inbound to v (if there are multiple ancestors inbound to v then s_a is the maximal $<^s$ precedence ancestor). Since $v = v_{l_1}^a = v_{l_1}^x$ then s_a is the parent of s_x (since if it is not then s_x 's parent would also have its first low point at v and so s_a would not be the maximal $<^s$ precedence segment).

The head edge of s_x is inserted into the cyclic order about v_h^x such that it is inserted adjacent to v_h^x 's parent edge (which is contained in s_a) and in the opposite cyclic direction to that of s_x . Following the boundary of the region containing the arc between v_h^x 's parent edge and e_h^x then the attachment to v is not performed relative to e_t^a but to the edges adjacent to e_t^a within the cyclic order around v at the instant when e_t^a was ordered.

Figure 4.4.4 gives two examples of this:

- Segment s_2 is embedded outside of segment s_1 (in an anti-clockwise direction) and both share a common tail vertex. The angles α and α' show the angles within the inside region (relative to s_2) adjacent to s_2 's head and tail vertices, respectively. s_2 's head edge is ordered clockwise of v_h^2 's parent and s_2 's tail edge is ordered anti-clockwise of the edge that was immediately clockwise of s_1 when s_1 was embedded (in this case v_t^1 's parent edge).
- Segment s_3 is embedded inside segment s_1 (in a clockwise direction) and β and β' show the angles within the inside region (relative to s_3) adjacent to s_3 's head and tail vertices, respectively. s_3 's head edge is ordered anti-clockwise (opposite to s_3 's cyclic direction) of v_h^3 's parent and s_3 's tail edge is ordered clockwise (same as s_3 's cyclic direction) of the edge that was immediately anti-clockwise (opposite to s_3 's cyclic direction) of s_1 when s_1 was embedded (in this case v_t^1 's stem edge).

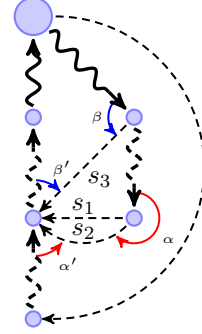


Figure 4.4.4: Angles From Parent & Neighbouring Edges To Segments With Increasing $<^s$ Precedence

This can be implemented such that:

DEFINITION 4.4.6. When a segment's tail edge is inserted into the cyclic edge order the pair of edges immediately clockwise and anti-clockwise can be added to a stack. When all descendants of that segment have been given a cyclic ordering then the entry can be removed from the tail of the stack.

If when a segment's tail edge is added to the cyclic order about a vertex that vertex contains no pairs of neighbouring edges on the stack then no ancestors have been connected to that vertex at their tail edges so the edge can be processed as per definition 4.4.5 and an entry placed on the stack containing the neighbouring edges.

If the vertex contains an entry on the stack then an ancestor has been connected to that vertex at its tail edge and the new segment's tail edge can be added to the cyclic order around that vertex such that if the new segment's cyclic direction is clockwise (anti-clockwise) then the tail edge is added to the cyclic order clockwise (anti-clockwise) from the edge on the stack that was anti-clockwise (clockwise) from the embedding of the previous ancestor.

Given this then a cyclic edge order can be generated for a biconnected graph.

Chapter 5

The Algorithm

This chapter takes the theoretical process defined over the previous two chapters and details how an efficient algorithm can be constructed. The algorithm is broken down into small sections corresponding to the sections in the previous chapters; these sections deal with:

- Expected Data Structures;
- Trémaux Tree Generation;
- Segment Generation;
- Planarity Testing and Permutation Generation;
- Permutation Manipulation; and
- Planar Embedding Generation.

The theory in the previous two chapters justifies why the algorithm works and as such:

- By performing a DFS and giving the vertices numbers, and edges direction, as they are visited then a Trémaux Tree is formed and has all the associated invariant properties thereof;
- By bucket sorting the edges according to low points (and an arbitrary order defined by the data structure used) then the edges are in $<^{**}$ order and a second DFS results in the segments being generated (and being in order of $<^s$); similarly, with all the associated invariant properties that this gives; then
- Given all these invariant properties then a simple iterative process to check segments for planarity/non-planarity and to embed segments can be used to generate an embedding.

Therefore, while the proof of the invariant properties relating to the Trémaux Tree, the segments and the segment's ordering is quite involved, the resulting algorithm can assume the existence of these properties through correct construction of the segments and becomes a relatively simple iterative process.

5.1 Depth First Search

LEMMA 5.1.1. Performing a Depth-First Search (DFS) on a graph defines a Trémaux Tree.

PROOF 5.1.2. Bondy and Murty [4, pg. 141-142] prove (using a timing function and the times each vertex is first and last visited) that the DFS-tree $\mathcal{T}$ of (multi-) graph $\mathcal{G}$ is such that every edge of $\mathcal{G}$ joins two vertices which are related (such that one vertex is contained in the path within the spanning tree from its root to the other vertex) in $\mathcal{T}$ – this is the fundamental definition of a Trémaux Tree (see page 28).

This proof is accepted with the caveat that Bondy and Murty do not consider looping edges; however, each looping edge has two related end-points (since each vertex is related to itself) so, trivially, is part of a Trémaux Tree. $\square$

Given lemma 5.1.1 then DFS is the fundamental building block of this planarity testing algorithm; in fact, each phase uses a separate DFS: first, to generate a Trémaux Tree; then on the (now directed) Trémaux Tree to generate the segments; thirdly on the segment's hierarchy to generate an embedding; and, after setting whether each block is *flipped* and the order of *permutable* segments, a final DFS on the segment's hierarchy to set the cyclic edge order.

Two equivalent implementations of a DFS algorithm are presented: one recursive (listing 5.1.1); the other iterative (listing 5.1.2). The recursive algorithm is slightly easier to comprehend and shorter; however, execution of the recursive version usually exceeds the memory allocated to the call stack (creating a stack overflow) long before either version would exceed the total available memory. As such, the iterative version is preferred but are interchangeable if memory issues are not of concern.

Two place holder functions `DFSPreOrder` and `DFSPostOrder` are included to represent when each implementation starts and finishes processing tree and cotree edges.

Listing 5.1.1: Recursive Depth-First Search

```

1 PROCEDURE RecursiveDFS( u ) {
2   MARK u AS VISITED
3   FOR EACH ( e  $\in$  v.connected ) {
4     IF ( e IS UNVISITED ) {
5       MARK e AS VISITED
6       LET e = {u,v}
7       IF ( v IS UNVISITED ) {
8         MARK e AS TREE EDGE
9         DFSPreOrder( e )
10        RecursiveDFS( v )
11        DFSPostOrder( e )
12      } ELSE {
13        MARK e AS COTREE EDGE
14        DFSPreOrder( e )
15        DFSPostOrder( e )
16      }
17    }
18  }
19 }
20 RecursiveDFS( root )

```

Listing 5.1.2: Iterative Depth-First Search

```

1 PROCEDURE IterativeDFS( root ) {
2   LET S = EMPTY STACK
3   MARK root AS VISITED
4
5   FOR EACH ( e  $\in$  root.connected [ IN
6     REVERSE ORDER ] )
7     PUSH e ONTO S
8
9   WHILE ( S IS NOT EMPTY ) {
10    peek at e on S
11    IF ( e IS UNVISITED ) {
12      MARK e AS VISITED
13      LET e = {u,v} WHERE ( v IS
14        UNVISITED OR v  $\preceq$  u )
15      IF ( v IS UNVISITED ) {
16        MARK v AS VISITED
17        MARK e AS TREE EDGE
18        DFSPreOrder( e )
19        FOR EACH ( e'  $\in$ 
20          v.connected \ {e} [ IN
21            REVERSE ORDER ] )
22          PUSH e' ONTO S
23        } ELSE {
24          MARK e AS COTREE EDGE
25          DFSPreOrder( e )
26          DFSPostOrder( e )
27          POP e FROM S
28        }
29      } ELSE {
30        IF ( e IS TREE EDGE )
31          DFSPostOrder( e )
32        POP e FROM S
33      }
34    }
35  }
36 }

```

Provided the place holder functions have a constant execution time then the DFS algorithm has an $O(\mathcal{V} + \mathcal{E})$ execution time [32].

5.2 Expected Data Structures

The algorithms listed in the following sections expect there to be data structures defined to represent a vertex and an edge. A template detailing the attributes, and respective initial values, is included below.

```

1 CLASS VERTEX {
2   ATTRIBUTE connected ← LIST OF edges PRE-POPULATED WITH ALL connected edges

4   ATTRIBUTE index      ← ( UNVISITED OR  $\mathbb{N}$  ) INITIALLY UNVISITED
5   ATTRIBUTE parent     ← ( EDGE OR NULL ) INITIALLY UNASSIGNED
6   ATTRIBUTE lowPoint1  ← VERTEX INITIALLY this
7   ATTRIBUTE lowPoint2  ← VERTEX INITIALLY this
8   ATTRIBUTE outbound   ← LIST OF edges INITIALLY EMPTY
9 }

```

A vertex is assumed to have a list of connected edges that is pre-populated prior to the planarity test and during pre-processing to generate the Trémaux Tree:

- Each vertex is assigned, as the DFS algorithm performs the search: a unique index, corresponding to the DFS pre-order; and, with the exception of the root vertex, a parent edge – which is the inbound tree edge via which that vertex was first reached by the DFS algorithm (the algorithm is started at the root vertex so in this special case the root vertex is defined to have a null parent).
- Each vertex calculates, as the DFS algorithm backtracks, its first and second low points.
- During bucket sorting, post DFS, those connected edges which have been assigned directionality outbound from that vertex are ordered and assigned to the outbound list.

```

10 CLASS EDGE {
11   ATTRIBUTE connected ← UNORDERED PAIR OF [PRE-POPULATED] INCIDENT VERTICES

13   ATTRIBUTE status    ← ( UNVISITED OR TREE OR COTREE ) INITIALLY UNVISITED
14   DERIVED ATTRIBUTE DFSFrom
15   DERIVED ATTRIBUTE DFSTo

17   METHOD setStatus( status ' ) {
18     status ← status '
19     IF ( status ' = UNVISITED ) { DFSFrom ← DFSTo ← UNASSIGNED }
20     ELSE {
21       LET {u,v} = connected WHERE u  $\preceq$  v
22       IF ( status ' = TREE ) { DFSFrom ← u, DFSTo ← v }
23       ELSE { DFSFrom ← v, DFSTo ← u }
24     }
25   }

27   METHOD otherEnd( v IN connected ) {
28     LET {x,y} ← connected
29     IF ( v = x ) RETURN y
30     ELSE RETURN x
31   }
32 }

```

Each edge is assumed to have been, similarly, initialised prior to planarity testing such that its connected attribute contains the (unordered) pair of vertices that edge joins. During DFS, each edge is assigned the status as either a tree or a cotree edge and from this, and the relative precedence of the incident vertices, a direction can be derived for that edge such that it is outbound from one incident vertex and inbound to the other.

5.3 Trémaux Tree Generation Algorithm

The first phase of the planarity testing is to pre-process the graph; this is performed by the depth-first search and subsequent bucket sort algorithm (listing 5.3.1); it is an equivalent non-recursive implementation to Hopcroft and Tarjan's [32, pgs. 553 & 555-556] recursive DFS and bucket sort.

The depth-first search is used to:

- Partition the edges into the sets of tree and cotree edges;
- During DFS pre-ordering, assign each edge a direction and, if a tree edge, assign the unvisited vertex a unique index corresponding to the DFS ascending pre-order;
- During DFS post-ordering, propagate the vertices' low points up the tree as the algorithm backtracks; and, using these low points, assigns edges to buckets.

After the DFS, the bucket sort then populates the list of outbound edges for each vertex.

Listing 5.3.1: Depth-First Search and Bucket Sort Pre-Processing Algorithm

```

1 PROCEDURE DFSandBucketSort(  $\mathcal{G}(\mathcal{V}, \mathcal{E})$ , root ) {
2   LET S = EMPTY STACK
3   LET bucket = ARRAY OF (2| $\mathcal{V}$ | - 1) EMPTY LISTS
4   LET i = 1
5   root.parent  $\leftarrow$  NULL
6   root.index  $\leftarrow$  i
7
8   FOR EACH ( e  $\in$  root.connected [IN REVERSE ORDER] )
9     PUSH e ONTO S
10
11  WHILE ( S IS NOT EMPTY ) {
12    PEEK AT e ON S
13    IF ( e.status = UNVISITED ) {
14      LET e = {u,v} WHERE ( v.index = UNVISITED OR v  $\preceq$  u )
15      IF ( v.index = UNVISITED ) {
16        v.index  $\leftarrow$  i  $\leftarrow$  i + 1
17        v.parent  $\leftarrow$  e
18        e.setStatus( TREE )
19        FOR EACH ( e'  $\in$  v.connected \ {e} [IN REVERSE ORDER] )
20          PUSH e' ONTO S
21      } ELSE {
22        e.setStatus( COTREE )
23        UpdateLowPoints( e )
24        AssignEdgeToBucket( e )
25        POP e FROM S
26      }
27    } ELSE {
28      IF ( e.status = TREE ) {
29        UpdateLowPoints( e )
30        AssignEdgeToBucket( e )
31      }
32      POP e FROM S
33    }
34  }
35  FOR ( b  $\leftarrow$  1 ... 2| $\mathcal{V}$ | - 1 )
36    FOR EACH ( e  $\in$  bucket(b) )
37      APPEND e TO TAIL OF e.DFSFrom.outbound
38  }
```

The procedures **UpdateLowPoints** and **AssignEdgeToBucket** are included on page 78. These two procedures correspond to the place holder function **DFSPostOrder** in listing 5.1.1 and the **DFSPreOrder** place holder function corresponds to lines 16-17.

The output of the algorithm is such that:

- Each vertex is processed, either:
 - Prior to starting the DFS search, if it is the root vertex; or
 - When the first edge incident to that vertex is processed, if it is a non-root vertex.

In both cases, processing the vertex:

- Assigns it an index (lines 6 & 16) higher than all previously assigned indexes (hence, giving it a unique index);
 - Assigns the incident edge to be the vertex's parent (line 17) or a null parent in the case of the root vertex (line 5); and
 - Adds all other incident edges to the stack $\mathcal{S}$ (lines 8-9 & 19-20) for subsequent processing (hence, all edges on $\mathcal{S}$ are incident to at least one visited vertex).
- When processing an edge connecting a visited to an unvisited vertex it is designated to be a tree edge; and the unvisited vertex is processed (as described above).

Since tree edges always connect to an unvisited vertex (which immediately is visited) the the sub-graph of tree edges forms a (rooted) spanning tree. Also, given that vertices are assigned successively higher unique indexes, any vertex u on the path of tree edges from the root vertex r to any other vertex v has relative precedence and indexes such that $(u \prec v \Rightarrow u.index < v.index)$ and $(u = v \Leftrightarrow u.index = v.index)$.

- When processing an edge connecting two visited vertices then it is designated to be a cotree edge. A cotree edge $(v \xrightarrow{c} u)$ appears on stack $\mathcal{S}$ twice before it is processed; once when vertex u is processed (and v is unvisited) and is added again when v is processed – if the first incidence was processed before the second was added to the stack then the edge would connect a visited to an unvisited vertex which contradicts the original assertion that the edge connects two visited vertices.

Considering the time complexity of the DFS algorithm:

- Each edge is added to stack $\mathcal{S}$ once if it is a tree edge or twice if it is a cotree edge.
- Each edge on stack $\mathcal{S}$ is processed:
 - Twice if it is a tree edge – once to assign it to be a tree edge (lines 15-21) and then again, during backtracking, when it is removed from the stack (lines 28-32); or
 - Once, for each time it appears on the stack, if it is a cotree edge – when it is assigned to be a cotree edge and removed from the stack (lines 21-26 & 32).
- Since at each iteration of the DFS algorithm, either:
 - An unvisited edge is designated to be a tree edge (and the connected unvisited vertex is processed adding all incident edges to the stack);
 - An unvisited edge is designated to be a cotree edge then has its low points updated and is added to a bucket before being removed from the stack;
 - A visited tree edge has its low points updated and is added to a bucket before being removed from the stack; or
 - A visited cotree edge is removed from the stack.

Therefore, provided the procedures **UpdateLowPoints** and **AssignEdgeToBucket** execute in constant time bound then the algorithm will terminate, after visiting all edges and vertices, in a finite $O(\mathcal{V} + \mathcal{E})$ time bound.

Considering the precedence and indexes of two vertices, u and v , after pre-processing:

LEMMA 5.3.1. $u = v \Leftrightarrow u.index = v.index$

PROOF 5.3.2. This follows directly from lines 15-16 of listing 5.3.1:

- Each vertex has a singular numerical index since each vertex is assigned an index when an incident edge is processed and that vertex is found to be unvisited; the process of assigning the vertex an index means that it is not unvisited and cannot be subsequently assigned another index; and
- No two vertices can have the same index since the global counter used to assign indexes is incremented with each assignment.

Therefore, each vertex has a unique index and each index is unique to a vertex, proving this lemma. $\square$

LEMMA 5.3.3. If vertex u is contained in the path of tree edges from the root vertex to (the distinct) vertex v then the index assigned by the DFS algorithm to u is less than the index assigned to v ($u \prec v \Rightarrow u.index < v.index$).

PROOF 5.3.4. When an edge is designated as a tree edge it connects from the visited vertex u (which is already assigned an index value) to an unvisited vertex v , which is not assigned an index value but is then immediately assigned (line 16) a value higher than the previous highest index value assigned an index value so $u.index < v.index$. By definition 3.1.1, given a tree edge ($u \xrightarrow{T} v$) then $u \prec v$; therefore, for any tree edge ($u \xrightarrow{T} v$) then $u \prec v$ and $u.index < v.index$.

Applying this recursively, then sequence of vertices $\langle r, v_1, v_2, \dots, v_{n-1}, v_n \rangle$ connected by tree edges ($(r \xrightarrow{T} v_1), (v_1 \xrightarrow{T} v_2), \dots, (v_{n-1} \xrightarrow{T} v_n)$) is such that $r \prec v_1 \prec v_2 \prec \dots \prec v_{n-1} \prec v_n$ and that $r.index < v_1.index < v_2.index < \dots < v_{n-1}.index < v_n.index$.

Therefore, if $u \prec v$ then $u.index < v.index$. $\square$

This result is particularly useful as provided that whenever vertices are compared they are not in different branches of the Trémaux Tree then their relative precedence can be determined by a numerical comparison of their associated indexes. Considering the results of chapter 4 then:

- Each edge connects two comparable ($\perp$) vertices u and v so, by comparing the assigned numeric indexes, the relative precedence (or equality) can be determined;
- Each segment is a chain of edges such that each of its vertices is comparable with each other of its vertices – thus the segment's head and tail vertices are comparable so their numerical indexes can be used to determine relative precedence and the segment's class;
- Each non-root segment's head and tail vertices (and the second low-point vertex of that segment's head edge) are comparable to that segment's parent's head vertex (and, if the parent is of Class 3 or 4, tail vertex) – thus whether a Class 4 segment is *flippable* or not can be determined by the relative precedence of the indexes corresponding to those vertices; and
- The $<^s$ precedence order is such that given two segments s_1 and s_2 where $s_1 <^s s_2$ and s_1 is a sibling of, or sibling's descendant of, s_2 then $v_h^2 \preceq v_h^1$ and that v_t^1 is comparable to v_h^2 and v_t^2 . By comparing the relative precedence of v_t^1, v_h^2 and v_t^2 then a determination can be made as to whether s_1 is *fully embedded*, *conflicts* or is outside of (and permits an embedding) relative to s_2 ; therefore, this determination can be, equivalently, made by comparing the indexes associated with those vertices.

Therefore, vertices' indexes can be used in later phases of the planarity test to determine a segment's status and its conflicts with other segments; so the process of determining relative precedence between two comparable precedence vertices can be achieved in $O(1)$ time.

Listing 5.3 updates each vertex's first and second low-points and propagates these from the leaves to the root vertex as the algorithm backtracks and is functionally identical to the snippet of Hopcroft and Tarjan's algorithm presented in [32, Pg. 555].

```

39 PROCEDURE UpdateLowPoints( e ) {
40   LET u = e.DFSFrom
41   LET v = e.DFSTo
42   IF ( e.status = COTREE ) {
43     IF ( v < u.lowPoint1 )
44       u.lowPoint1 ← v
45     ELSEIF ( u.lowPoint1 < v < u.lowPoint2 )
46       u.lowPoint2 ← v
47   } ELSEIF ( e.status = TREE ) {
48     IF ( v.lowPoint1 < u.lowPoint1 ) {
49       IF ( v.lowPoint2 < u.lowPoint1 )
50         u.lowPoint2 ← v.lowPoint2
51     ELSE
52       u.lowPoint2 ← u.lowPoint1
53       u.lowPoint1 ← v.lowPoint1
54     } ELSEIF ( v.lowPoint1 = u.lowPoint1 ) {
55       IF ( v.lowPoint2 < u.lowPoint2 )
56         u.lowPoint2 ← v.lowPoint2
57     } ELSEIF ( v.lowPoint1 < u.lowPoint2 )
58       u.lowPoint2 ← v.lowPoint1
59   }
60 }
```

Given lemmas 5.3.1 & 5.3.3 then **UpdateLowPoints** can be implemented by comparing the numerical indexes for each vertex in question and therefore, as a series of simple comparisons and assignments, can be executed in a constant time bound.

Listing 5.3 orders the edges by $\prec^*$ such that if two edges outbound from a common vertex have incomparable ($\parallel^*$) precedence they are assigned to the same bucket and their order within the bucket is given by the order they are processed by the DFS algorithm. This procedure is the same as the function ϕ used by Hopcroft and Tarjan [32, Pg. 556].

```

61 PROCEDURE AssignEdgeToBucket( e ) {
62   IF ( e.status = COTREE )
63     APPEND e TO TAIL OF bucket( 2 × e.DFSTo.index - 1 )
64   ELSEIF ( e.DFSTo.lowPoint2 < e.DFSFrom )
65     APPEND e TO TAIL OF bucket( 2 × e.DFSTo.lowPoint1.index )
66   ELSE
67     APPEND e TO TAIL OF bucket( 2 × e.DFSTo.lowPoint1.index - 1 )
68 }
```

Each bucket is a list and since elements can be appended to a list in constant time (and the rest of the procedure is simple comparisons and calculations) then the procedure can be executed in a constant time bound.

Therefore, since these two procedures can be executed in constant time and the DFS portion of this phase executes in an $O(\mathcal{V} + \mathcal{E})$ time bound if they are constant then the DFS search has a linear time bound.

Considering the bucket sort, there are $O(\mathcal{V})$ buckets that must each be checked and, in total, each edge is added to exactly one bucket so there will be $O(\mathcal{E})$ to be iterated through; therefore, the bucket sort also has an $O(\mathcal{V} + \mathcal{E})$ time bound and so does this entire phase of the algorithm.

5.4 Segment Generation Algorithm

The segment class requires an extra attribute in each of the vertex and edge classes to store a reference to the segment containing that element.

```

1 CLASS Vertex {
2   ...
3   ATTRIBUTE segment ← NULL
4 }

```

```

5 CLASS Edge {
6   ...
7   ATTRIBUTE segment ← NULL
8 }

```

The segment class is a list of alternating vertices and edges (although it is possible to define a segment purely as a list of edges and obtain the connected vertices as required) associated with a lists of child segments for each Class of segment.

A global list of segments is maintained to store the segments in the order they are generated in (which corresponds to the $<^s$ order). Whenever a new segment is created it is automatically added to the global list by the segment's constructor (this assumes that only multiple planarity tests are not performed concurrently) and to the appropriate list within its parent based on that segment's class.

```

9 GLOBAL SegmentList = EMPTY LIST

11 CLASS Segment {
12   ATTRIBUTE path ← EMPTY LIST
13   ATTRIBUTE class1children ← EMPTY LIST
14   ATTRIBUTE class2children ← EMPTY LIST
15   ATTRIBUTE class3children ← EMPTY LIST
16   ATTRIBUTE class4children ← EMPTY LIST

18   CONSTRUCTOR Segment( x ) {
19     this.path ← LIST CONTAINING x
20     x.segment ← this
21     IF ( getParent() ≠ NULL ) {
22       SWITCH ( getSegmentClass() ) {
23         CASE 1: APPEND this TO getParent().class1children, BREAK
24         CASE 2: APPEND this TO getParent().class2children, BREAK
25         CASE 3: APPEND this TO getParent().class3children, BREAK
26         CASE 4: APPEND this TO getParent().class4children
27       }
28     }
29     APPEND this TO TAIL OF SegmentList
30   }

32   METHOD addElement( x ) {
33     ASSERT( this.canBeExtendedBy( x ) )
34     APPEND x TO TAIL OF this.path
35     x.segment ← this
36   }

38   METHOD getHead() { RETURN this.path.getHead() }
39   METHOD getTail() { RETURN this.path.getTail() }

41   METHOD getHeadEdge() {
42     IF ( getHead() IS EDGE ) RETURN getHead()
43     ELSE RETURN getHead().outbound.getHead()
44   }

46   METHOD getTailEdge() {
47     IF ( getTail() IS EDGE ) RETURN getTail()
48     ELSE RETURN getTail().parent
49   }

```

```

51  METHOD getHeadVertex() {
52      IF ( getHead() IS VERTEX ) RETURN getHead()
53      ELSE RETURN getHead().DFSFrom
54  }

56  METHOD getTailVertex() {
57      IF ( getTail() IS VERTEX ) RETURN getTail()
58      ELSE RETURN getTail().DFSTo
59  }

61  METHOD getSegmentClass() {
62      IF ( getTail() IS VERTEX ) RETURN 1
63      ELSE IF ( getHeadVertex() < getTailVertex ) RETURN 2
64      ELSE IF ( getHeadVertex() = getTailVertex ) RETURN 3
65      ELSE RETURN 4
66  }

68  METHOD getParent() {
69      IF ( getHead() IS VERTEX ) RETURN NULL
70      ELSE RETURN getHead().DFSFrom.segment
71  }

73  METHOD canBeExtendedBy( x ) {
74      IF ( x IS EDGE ) RETURN ( getTail() = x.DFSFrom )
75      ELSE RETURN ( getTail() IS TREE EDGE AND getTail().DFSTo = x )
76  }
77  }

```

The procedure to generate segments performs a simple DFS over the outbound edges (which are in $<^*$ order from the bucket sort) from each vertex; as each edge is considered by the DFS it either extends the existing segment's path (and so is added to that path) or is the head of a new segment. The segments are stored in the global list, as previously mentioned; as well as being accessible via the segment attribute of each vertex and edge and from other segments via the parent-child hierarchy.

Listing 5.4.1: Segment Generation Algorithm

```

78  PROCEDURE GenerateSegments( root ) {
79      LET E = EMPTY STACK
80      LET s = NEW Segment( root )

82      FOR EACH ( e ∈ root.outbound [IN REVERSE ORDER] )
83          PUSH e ONTO E

85      WHILE ( E IS NOT EMPTY ) {
86          POP e FROM E
87          IF ( s = NULL ) LET s = NEW Segment( e )
88          ELSE s.addElement( e )

90          IF ( e.status = COTREE ) {
91              LET s = NULL
92          } ELSE {
93              LET v = e.DFSTo
94              s.addElement( v )
95              IF ( v.outbound IS EMPTY )
96                  LET s = NULL
97              ELSE
98                  FOR EACH ( e' ∈ v.outbound [IN REVERSE ORDER] )
99                      PUSH e' ONTO E
100          }
101      }
102  }

```

When a given vertex is processed then its outbound edges are added to the top of the stack in reverse order (such that the maximum $<^{**}$ outbound edge is added first and the minimum $<^{**}$ outbound edge is added last and is on the top of the stack); thus when the next edge is processed (from the top of the stack) it will be the minimum $<^{**}$ precedence edge outbound from the last vertex and will correctly extend the segment containing that given vertex. The algorithm can then check to see if a leaf has been reached (i.e. a cotree edge or a vertex with no outbound edges) and if so sets the current segment to null indicating that when the next edge (if any) is processed then the algorithm will have backtracked and be starting a new segment (with the next minimum $<^{**}$ precedence outbound edge from the most recent, highest $<$ precedence, vertex still with unprocessed outbound edges). Therefore, not only does the algorithm generate the segments simply; it also generates them in $<^s$ order.

Since each edge is only outbound from one vertex then each edge is added to stack E once so can only be removed from E once and added to a segment once. Similarly, each vertex is added to a segment if it is the root vertex or when its parent tree edge is processed and so each vertex is only processed once and added to a single segment. Given this then segment generation is $O(\mathcal{V} + \mathcal{E})$

Listing 5.4.1 is very similar to the pathfinder procedure of Hopcroft and Tarjan [32, Pg. 556] – the fundamental difference is that where H&T assume a biconnected graph they can also assume that every segment is terminated by a cotree edge and as soon as a cotree edge is found the old segment can be closed and a new one started. This algorithm also handles the case of Class 1 segments where the segment's tail is a vertex with no outbound edges.

5.5 Planarity Testing and Permutation Generation Algorithm

Prior to the implementation of the planarity test, some useful attributes and methods can be added to the segment class.

- The `lastEmbeddedSegment` attribute is a reference to the most recently embedded child segment of this parent and is used to determine whether the currently embedded segment (also a child of this parent) is *permutable* with that most recently embedded child (its sibling).
- The *blocks* attribute is simply that – a stack of blocks which are relative to this segment. It is used as a temporary holder for blocks during embedding such that when all of a segment's descendants are embedded the method `appendDescendantSegmentsToParent` can be invoked and all blocks of *partially embedded* segments can then be *fully embedded* if they are on the outside (and ignored) or else, if on the inside, appended to the block containing this segment's parent.
- The permutation attribute is a reference to a permutable group of segments – initially, the segment does not belong to any such group but may be assigned to one if, when compared to the `lastEmbeddedSegment` of its parent it is found to be permutable.
- The embedded attribute is a boolean flag indicating whether the segment was successfully embedded or not.

Listing 5.5.1: Extending the Segment Class

```

1  CLASS Segment {
2
3      ...
4
5      ATTRIBUTE lastEmbeddedSegment ← NULL
6      ATTRIBUTE blocks                ← EMPTY STACK
7      ATTRIBUTE permutation           ← NULL
8      ATTRIBUTE embedded              ← FALSE
9
10     METHOD getLowPoint1() {
11         IF ( getHead() IS VERTEX )      RETURN getHead();
12         IF ( getHead().status = COTREE ) RETURN getHeadVertex();
13         RETURN min( getHead().DFSTo.lowPoint1, getHeadVertex() )
14     }
15
16     METHOD getLowPoint2() {
17         IF ( getHead() IS VERTEX )      RETURN getHead();
18         IF ( getHead().status = COTREE ) RETURN getHeadVertex();
19         RETURN min( getHead().DFSTo.lowPoint2, getHeadVertex() )
20     }
21
22     METHOD hasBridgeEdges() { RETURN getSegmentClass ≤ 2 }
23     METHOD isRotatable()    { RETURN NOT isStatic() AND getSegmentClass ≤ 3 }
24     METHOD isReversible()   { RETURN NOT getParent() = NULL AND 2 ≤
                             getSegmentClass ≤ 3 }
25
26     METHOD isStatic() {
27         RETURN ( getParent() = NULL )
28         OR ( getParent().getParent() = NULL AND this =
              getParent().children.getHead() )
29     }
30
31     METHOD isFlippable() {
32         RETURN NOT isStatic()
33         AND getSegmentClass() = 4
34         AND ( ( getParent().getHeadVertex() ≪ getTailVertex() )
35              OR ( getParent().getTailVertex() = getTailVertex() )

```



```

36         AND getParent().getHeadVertex()  $\preceq$  getLowPoint2Vertex() ) )
37     }
38
39     METHOD conflictsWith( s ) {
40         RETURN ( s.getTailVertex()  $\prec$  this.getTailVertex()
41              $\prec$  s.getHeadVertex()  $\prec$  this.getHeadVertex() )
42             OR ( this.getTailVertex()  $\prec$  s.getTailVertex()
43                  $\prec$  this.getHeadVertex()  $\prec$  s.getHeadVertex() )
44     }
45
46     METHOD appendDescendantSegmentsToParent() {
47         IF ( getParent() = NULL ) RETURN
48         LET tempSegmentList = EMPTY LIST
49         WHILE ( NOT blocks.isEmpty() ) {
50             REMOVE b FROM TAIL OF blocks
51             IF ( NOT b.isFlippable() )
52                 MOVE ALL FROM b.partialInside TO HEAD OF tempSegmentList
53         }
54         getParent().blocks.getTail().appendSegmentsAfter( this , tempSegmentList )
55     }
56 }

```

Considering the time complexity of the methods added to the the segment class:

- The method **appendDescendantSegmentsToParent** iterates through every block containing non-*flippable partially embedded* descendants (relative to that segment) on its inside and moves them out of those blocks and into the block containing this segment (and on the same side as this segment). It performs this operation by moving the *partially embedded* contents of the block to a temporary list (so as to preserve their order) and then moving the entire temporary list to the parent block. Each movement of the list can be performed by disconnecting the list elements from its containing data structure and connecting it into the new list such that only the head and tail of the list needs considering and so can be performed in a constant time bound.

If there are b blocks of non-*flippable partially embedded* segments then each block is removed from the list of blocks once and can be added to the temporary list, at most, once which all takes $O(b)$ time; the temporary list can then be appended to the parent's block which takes $O(1)$ time. Therefore, the method takes $O(b)$ time.

- All the other methods perform simple comparisons and execute in a constant time bound.

A block contains four lists: the *partially embedded* segments inside and outside of the block; and a list of all segments (*partially* or *fully embedded*) that have been added directly to that block (not indirectly when moved from a descendant).

It also contains attributes which are:

- A reference to the previous block on the same parent (forming a singly linked list) that is used when concatenating blocks;
- A reference to the parent segment that the block is relative to; and
- A boolean flag containing the status as to whether all segments of the block (*partially* or *fully embedded*) are flippable – if so then the block is flippable.

Most of the methods are self explanatory, however:

- `fullyEmbedTo` takes a vertex as an argument and checks the *partially embedded* segments to see if, relative to that vertex, they are now *fully embedded* and have no further influence on future segment's embeddings. Each time a segment is found it is removed from the list of *partially embedded* segments; the method returns true if all the block's segments are *fully embedded* and false if there exists at least one *partially embedded* segment. If there are x segments currently with the status *partially embedded* that change to *fully embedded* then the method has a $O(x)$ upper time bound.
- The `concatenate` method coalesces the block with its immediate predecessor; this can be achieved in a constant time bound by manipulating the references to the head and tail of the list rather than iterating over the entire list.
- All other methods are self explanatory and execute in a constant upper time bound.

Listing 5.5.2: Block Class

```

57 GLOBAL BlockList = EMPTY LIST

59 CLASS Block {
60     ATTRIBUTE partialInside    ← EMPTY LIST
61     ATTRIBUTE partialOutside   ← EMPTY LIST
62     ATTRIBUTE allInside        ← EMPTY LIST
63     ATTRIBUTE allOutside       ← EMPTY LIST
64     ATTRIBUTE previous
65     ATTRIBUTE parent
66     ATTRIBUTE flippable

68     CONSTRUCTOR Block( s ) {
69         APPEND s TO TAIL OF this.partialInside
70         APPEND s TO TAIL OF this.allInside
71         this.parent      ← s.getParent()
72         this.previous     ← parent.blocks.getTail()
73         this.flippable    ← s.isFlippable()
74         this.parent.lastEmbeddedSegment ← s
75         APPEND this TO TAIL OF parent.blocks
76         APPEND this TO TAIL OF BlockList
77     }

79     METHOD addSegmentInside( s ) {
80         APPEND s TO TAIL OF this.partialInside
81         APPEND s TO TAIL OF this.allInside
82         IF ( this.flippable ) this.flippable = s.isFlippable()
83         this.parent.lastEmbeddedSegment = s
84     }

86     METHOD addSegmentOutside( s ) {
87         APPEND s TO TAIL OF this.partialOutside
88         APPEND s TO TAIL OF this.allOutside
89         this.parent.lastEmbeddedSegment = s
90     }

```

```

92  METHOD appendSegmentsAfter( s, SList ) {
93      IF ( this.partialInside.getTail() = s )
94          APPEND SList TO TAIL OF this.partialInside
95      ELSE APPEND SList TO TAIL OF this.partialOutside
96  }

98  METHOD fullyEmbedTo( v ) {
99      WHILE ( NOT this.partialInside.isEmpty()
100          AND v  $\preceq$  this.partialInside.getTail().getTailVertex() )
101          REMOVE TAIL FROM this.partialInside
102      WHILE ( NOT this.partialOutside.isEmpty()
103          AND v  $\preceq$  this.partialOutside.getTail().getTailVertex() )
104          REMOVE TAIL FROM this.partialOutside
105      RETURN ( this.partialInside.isEmpty() AND this.partialOutside.isEmpty() )
106  }

108  METHOD conflictInsideWith( s ) {
109      RETURN NOT this.partialInside.isEmpty()
110          AND this.partialInside.getTail().conflictsWith( s )
111  }

113  METHOD conflictOutsideWith( s ) {
114      RETURN NOT this.partialOutside.isEmpty()
115          AND this.partialOutside.getTail().conflictsWith( s )
116  }

118  METHOD concatenate() {
119      this.previous.flippable  $\leftarrow$  ( this.previous.flippable AND this.flippable )
120      MOVE ALL FROM this.partialInside TO TAIL OF this.previous.partialInside
121      MOVE ALL FROM this.partialOutside TO TAIL OF this.previous.partialOutside
122      MOVE ALL FROM this.allInside TO TAIL OF this.previous.allInside
123      MOVE ALL FROM this.allOutside TO TAIL OF this.previous.allOutside
124      REMOVE this FROM TAIL OF this.parent.blocks
125  }

127  METHOD flip() {
128      SWAP partialInside AND partialOutside
129      SWAP allInside AND allOutside
130  }
131 }

```

The procedure `checkEmbedding` is used to check a segment and a block to see if they are compatible for an embedding. Given that the segment can be tested against the inside and outside of the block for a conflict in a constant time and the rest of the method is simple comparisons then the procedure has an $O(1)$ upper time bound.

There are six possible outcomes:

1. When the segment conflicts with both sides of the block and is non-planar;
2. When the segment conflicts with neither side of the block and so can form a new block inside the previous;
3. When the segment does not conflict with the inside of the block but does conflict with the outside – so can, potentially, be embedded directly onto the inside of the block;
4. When the segment conflicts on the inside but not the outside and the segment is *flippable* – so can, potentially, be embedded directly onto the outside of the block;
5. When the segment conflicts on the inside but not the outside and the segment is not *flippable* but the block is – then the block must be *flipped* for an embedding to occur; and
6. When the segment conflicts on the inside but neither segment nor block can be *flipped* and so the graph is non-planar.

```

132 PROCEDURE checkEmbedding( segment, block ) {
133   LET inConflict    = block.conflictInsideWith( segment )
134   LET outConflict   = block.conflictOutsideWith( segment )
135   IF ( inConflict AND outConflict )           RETURN CONFLICT_ON_BOTH_SIDES
136   IF ( NOT inConflict AND NOT outConflict ) RETURN NEW_BLOCK
137   IF ( NOT inConflict )                       RETURN EMBED_INSIDE
138   IF ( segment.isFlippable() )                 RETURN EMBED_OUTSIDE
139   IF ( block.isFlippable() )                   RETURN FLIP_EMBED_INSIDE
140   RETURN CONFLICT_INSIDE
141 }
```

Below is the planarity testing algorithm. It uses a DFS over the segment hierarchy – on the forward search (lines 162-242) the algorithm tests each segment against the most recent block(s) relative to its parent and on the backtracking phase (lines 244-245) it propagates the *partially embedded* descendant segments from their current block to the parent block.

```

142 PROCEDURE AddChildrenToStack( segment, S ) {
143   FOR ( s ∈ segment.class4children [IN REVERSE ORDER] )
144     PUSH s ONTO S
145   FOR ( s ∈ segment.class3children [IN REVERSE ORDER] )
146     PUSH s ONTO S
147   FOR ( s ∈ segment.class2children [IN REVERSE ORDER] )
148     PUSH s ONTO S
149   FOR ( s ∈ segment.class1children [IN REVERSE ORDER] )
150     PUSH s ONTO S
151 }

153 PROCEDURE TestPlanarity( rootVertex ) {
154   LET lastSegmentProcessed ← NULL
155   LET S ← EMPTY STACK

157   AddChildrenToStack( rootVertex.segment, S )

159   WHILE ( S IS NOT EMPTY ) {
160     PEEK AT s ON S
161     IF ( NOT s.embedded ) {
162       IF ( s.getSegmentClass() = 4 ) {
163         LET p ← s.parent
164         LET b ← p.blocks.getTail()
165         LET l ← p.lastEmbeddedSegment

167         IF ( s AND l ARE PERMUTABLE ) {
168           IF ( l.permutation IS NOT NULL ) {
169             APPEND s TO TAIL OF l.permutation
170             s.permutation ← l.permutation
171           } ELSE
172             s.permutation ← l.permutation ← LIST CONTAINING l AND s
173             b.appendSegmentsAfter( l, s )
174             p.lastEmbeddedSegment ← s
175         } ELSE {
176           WHILE ( b IS NOT NULL ) {
177             IF ( NOT b.fullyEmbedTo( s.getHeadVertex() ) )
178               BREAK
179             s.blocks.removeTail()
180             b ← p.blocks.getTail()
181           }

183           IF ( b IS NULL ) {
184             NEW Block( s )
185           } ELSE {
186             LET side ← INSIDE
187             LET EMPTY ← FALSE
188             LET done ← FALSE
189             SWITCH ( checkEmbedding( s, b ) ) {
190               CASE CONFLICT_BOTH_SIDES: RETURN NON_PLANAR
191               CASE CONFLICT_INSIDE:      RETURN NON_PLANAR
192               CASE NEW_BLOCK:           NEW Block( s )
193                                         CONTINUE
194               CASE EMBED_OUTSIDE:      side ← OUTSIDE
195                                         EMPTY ← b.isEmptyOutside()
196                                         CONTINUE
197               CASE FLIP_EMBED_INSIDE:  b.flip();
198               CASE EMBED_INSIDE:       EMPTY ← b.isEmptyInside()
199             }
200             WHILE ( EMPTY AND b.previous IS NOT NULL ) {

```

```

201         EMPTY ← FALSE
202         LET pside ← INSIDE
203         SWITCH ( checkEmbedding( s, b.previous ) ) {
204         CASE CONFLICT_BOTH_SIDES: RETURN NON_PLANAR
205         CASE CONFLICT_INSIDE: RETURN NON_PLANAR
206         CASE NEW_BLOCK: IF ( side = INSIDE )
207                         b.addSegmentInside( s )
208                         ELSE
209                         b.addSegmentOutside( s )
210                         done = TRUE
211                         CONTINUE
212         CASE EMBED_OUTSIDE: pside ← OUTSIDE
213                             EMPTY ←
214                             b.previous.isEmptyOutside()
215                             CONTINUE
216         CASE FLIP_EMBED_INSIDE: b.previous.flip();
217         CASE EMBED_INSIDE: b.previous.isEmptyInside()
218                             EMPTY ←
219                             IF ( NOT done ) {
220                                 IF ( pside ≠ side ) {
221                                     IF ( b.previous.isFlippable() )
222                                         b.previous.flip()
223                                     ELSE IF ( b.isFlippable() ) {
224                                         b.flip()
225                                         side ← pside
226                                     } ELSE
227                                     RETURN NON_PLANAR
228                                 }
229                                 b.concatenate()
230                                 b ← b.previous
231                             }
232         IF ( NOT done ) {
233             IF ( side = INSIDE )
234                 b.addSegmentInside( s )
235             ELSE
236                 b.addSegmentOutside( s )
237         }
238     }
239 }
240 }
241 AddChildrenToStack( s, S )
242 s.embedded ← TRUE
243 } ELSE {
244     s.appendDescendantSegmentsToParent()
245     POP s FROM S
246 }
247 }
248 }

```

Looking at the algorithm in more depth:

- Lines 167-174 consider when a segment s and its preceding sibling l are permutable; which occurs when $v_t^s = v_t^l$ and $v_h^s = v_{l_2}^s = v_h^l = v_{l_2}^l$.

In this case, the new segment and the old are added to a *permutable* group – this can initially be considered to be a linked list structure for its ability to be populated in an upper time bound equal to the number of elements added to the list (without knowing the size of the group previously). Thus p permutable segments can be embedded in $O(p)$ time.

- Lines 176-181 consider whether the *partially embedded* segments are *fully embedded*. If

they are then they are removed from the block and if the block is emptied then it too is removed; thus leaving the stack of blocks either with only *partially embedded* segments or empty.

The time complexity of this part is considered later with the concatenation of blocks.

- Line 184 is reached either by the first child segment or when all previous siblings and their descendants have been *fully embedded* – in this case the segment has a trivial embedding inside its parent and forms a new block.

This can be achieved in a constant upper time bound.

- Lines 186-199 tests the segment for an embedding against the last block. Since it is constant upper time bound to check the status of a segment's embedding with a block and to flip blocks and add segments to blocks then this portion of the algorithm can be performed in constant time.
- Lines 201-217 performs a similar test on the penultimate block. Again requiring a constant upper time bound for this portion.
- However, considering the entire while loop (lines 200-230) then the result of one iteration is that either:
 - The case NEW_BLOCK is reached and the segment is added to the last block without affecting the penultimate block and the loop terminates;
 - A NON_PLANARITY case is reached and the entire procedure terminates; or
 - The last and penultimate blocks are concatenated.

In the first two cases the algorithm breaks out of the loop and can perform all actions within that iteration of the loop in constant time. In the last case, the algorithm iterates through the loop and performs a concatenation between the last and penultimate blocks – this is achieved in constant time bound but if the both the last and penultimate blocks are empty on the side in which the segment is being embedded then the loop can iterate again and a new penultimate block can be considered.

Considering the algorithm in totality, then the individual portions of the algorithm have constant execution time within a single iteration with the exception of: looping to remove *fully embedded* segments and empty blocks; looping to concatenate blocks; and looping during backtracking to move segments to a parent block (and discard the child blocks).

In all these cases either: a segment changes status from *partially* to *fully embedded*; and/or a block is discarded. Since each segment can only have this status change once and that each block can only be emptied and discarded once then this may occur multiple times within a single iteration but there is a finite number of times it can occur over all iteration – such that at most there can be $O(S)$ times at which a segment is removed from a block and $O(S)$ times a block can be deleted.

Therefore, if there are S segments then each iteration takes a constant time plus there is additional $O(S)$ time spread across all iterations so the total execution time for the planarity test is $O(S)$.

Each segment is *partially embedded* within a single block and can only be *fully embedded* once, removing it from that block. This occurs either when testing a subsequent segment for embedding or during backtracking but can only occur in one of these cases for each segment.

Similarly, each block can only be formed once and can be discarded if either: all segments within it are *fully embedded* or if it is concatenated with a previous block.

5.6 Reordering Permutations

5.6.1 Steinhaus-Johnson-Trotter Permutation Algorithm

The Steinhaus-Johnson-Trotter algorithm is a method of generating permutations such that *"each permutation is derived from its predecessor by the interchange of two marks in adjacent positions. Moreover, the rules are extremely simple"* [38, pg. 282].

The algorithm was originally discovered by Hugo Steinhaus (Summarised in [40, Section 3.4]) and rediscovered independently by many others including Trotter [76] and Johnson [38].

The algorithm is summarised below:

- The elements of the input sequence are each assigned an index (a unique sequential value) of ascending value corresponding to the element's position in the sequence (lowest index at the head of the list);
- Elements are given a direction of travel; initially towards the head of the sequence;
- Elements are mobile if, in their direction of travel, they are not at the end of the list or the adjacent element has a higher index;
- At each stage, the mobile element with the highest index is swapped with the adjacent element in its current direction of travel and the direction of travel is reversed for all (immobile) elements with a higher index
- The result of that stage is a new permutation which differs from the permutation at the previous stage by the transposition of a pair of adjacent elements.

This algorithm follows a predictable iterative pattern for a sequence of size N :

- The largest indexed element is moved from the tail-to-head of the sequence in $(N - 1)$ swaps;
- On the N^{th} swap the element with the largest index is immobile at the head of the sequence and the sub-sequence of lower indexed elements is considered and one step of the algorithm is performed on this sub-sequence, moving the next highest indexed mobile element;
- On the $(N + 1)^{\text{th}}$ to $(2N - 1)^{\text{th}}$ swaps the largest element is moved head-to-tail of the sequence;
- On the $2N^{\text{th}}$ swap, when the element with the largest index is immobile at the tail of the sequence, again the sub-sequence is considered and the next highest indexed mobile element is moved;
- This process is repeated, in batches of $2N$ swaps, until all permutations of the sub-sequence of lower indexed elements has been cycled through (in $N!$ steps).
- When the algorithm terminates no elements will be mobile (element 2 will be at the head of the sequence followed by element 1, both of which will have direction towards the head, and elements 3 to N will be in that order and have direction towards the tail) and the algorithm can be reset by swapping the lowest and second lowest indexed elements and resetting the element's direction of travel to be towards the head.

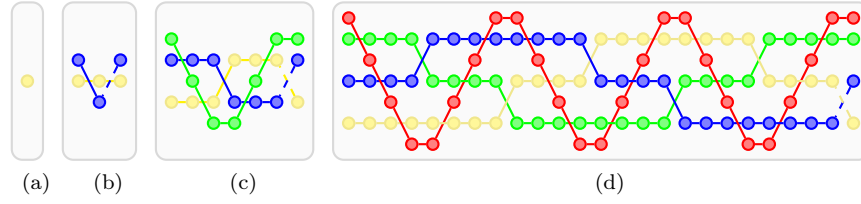


Figure 5.6.1: Graphical Representation of Steinhaus-Johnson-Trotter Permutation Algorithm for One (a) to Four (d) Elements Showing how the Pattern of Permutations Generated Builds on the Previous.

Figure 5.6.1 shows the sequential swaps performed by the algorithm for sets of size one (a) through four (d). Each permutation is a vertical group of elements with: the initial state on the left; the terminating state (where no elements are mobile) as the penultimate vertical group on the right; and the swap (dashed lines) required to reset the sequence's order on the right. It illustrates the relationship between (previous) permutations of a set of elements and (next) permutations for that set with an additional element such that successive sub-figures show the movement of that additional element and when that additional element reaches an outermost position the next swap of the previous permutation is performed.

Simplified Steinhaus-Johnson-Trotter Permutation Algorithm

As previously noted, the Steinhaus-Johnson-Trotter algorithm follows a repeating pattern of transpositions for the highest indexed element and this pattern can be iteratively applied to the sub-sequence of lower indexed elements. As a result the element and swap direction can be calculated:

- For a set with N elements, when cycling through permutations, the largest (N^{th}) indexed element (initially at the tail of the order) will be moved when the number (C) of swaps, from the initial order, (designated as the Current Permutation Index) is not exactly divisible by N (i.e. $\text{modulo}(C, N) \neq 0$) and the direction of travel will be towards the head of the sequence if $(\text{modulo}(C, 2N) < N)$ or towards the tail otherwise. Therefore, for out of every N steps (or $(N! - (N - 1)!)$ out of $N!$ total permutations) the element with the largest indexed element will be moved.
- The element with the second largest index will move when $(\text{modulo}(C, N) = 0)$ and $(\text{modulo}(\frac{C}{N}, N - 1) \neq 0)$ and the direction of travel is towards the head when $(\text{modulo}(\frac{C}{N}, 2(N - 1)) < (N - 1))$ or towards the tail otherwise. Therefore for every $(N - 2)$ out of $N(N - 1)$ steps (or $((N - 1)! - (N - 2)!)$ out of $N!$ total permutations) the second largest indexed element will be moved.
- In general, the K^{th} indexed element of an N element set will move if C is exactly divisible by $(\frac{N!}{K!})$ but is not exactly divisible by K and its direction of travel is towards the head if $(\text{modulo}(C \frac{K!}{N!}, 2K) < K)$ or towards the tail otherwise. The K^{th} element will be swapped $(K! - (K - 1)!)$ out of $N!$ total permutations.

Given this general formula and the current permutation index for the permutation it is possible to calculate which element will be moved and its direction of travel. Testing the elements of the sequence to locate the position of the highest indexed mobile element has an $O(N)$ upper time bound (where N is the number of elements in the sequence). This time bound

is identical to the time bound when testing the mobility of element of successively decreasing indexes against their neighbour in their direction of travel; however, there is no requirement to store the direction of travel as this can be calculated at the same time as the index of the highest mobile element is found nor is there a requirement to update the direction of travel after performing a transposition. It also allows for a simple algorithm to jump to a specific permutation index (see sub-section 5.6.1 on page 95).

The following describes the data structures required:

- *permutation* : $\langle X \rangle$ A permutable sequence (in this case, the elements contained are segments) provided as input with a static order. This is assumed to be pre-generated prior to initialising indexes and performing transpositions.
- *elementIndex* : $\mathbb{N} \rightarrow \mathbb{N}$ A mapping from the index of an element within the original sequence to the current position of that element within the current permutation.
- *elementPosition* : $\mathbb{N} \rightarrow \mathbb{N}$ A mapping from the current position of an element to the index of that element. This is the inverse mapping of *elementIndex* and is used to reference adjacent elements during transposition.
- *currentPermutation* : $\mathbb{N}$ The current permutation index (how many transpositions since the original).
- *totalPermutations* : $\mathbb{N}$ The total number of permutations for the sequence; a constant value equal to the factorial of the size of the permutable sequence.

The input permutation is assumed to be pre-generated but the other data can be initialised as follows:

```

1 currentPermutation ← 0
2 totalPermutations ← 1
3 FOR ( i ← 1 ... |permutation| ) {
4     elementIndex(i) ← i
5     elementPosition(i) ← i
6     totalPermutations ← totalPermutations × i
7 }
```

A simple procedure is required to swap the position of two adjacent elements and takes two arguments: the index of the current mobile element and a boolean flag for the direction of travel (TRUE indicates towards the head and FALSE towards the tail).

```

8 PROCEDURE swapElement( mobileElement, direction ) {
9     mobilePosition ← elementPosition( mobileElement )
10    IF ( direction ) swapPosition ← mobilePosition - 1
11    ELSE swapPosition ← mobilePosition + 1
12    swapElement ← elementIndex( swapPosition )
13    elementPosition( mobileElement ) ← swapPosition
14    elementPosition( swapElement ) ← mobilePosition
15    elementIndex( mobilePosition ) ← swapElement
16    elementIndex( swapPosition ) ← mobileElement
17 }
```

The result is that the element indexes at adjacent positions are swapped and the element positions of those indexes are swapped.

The procedure to step from one permutation to the next is:

```

18 PROCEDURE nextPermutation() {
19   p ← currentPermutation ← modulo(currentPermutation + 1, totalPermutations)
20   FOR ( m ← |permutation| ... 2 ) {
21     IF ( modulo( p, m ) ≠ 0 ) {
22       IF ( modulo( p, 2 × m ) < m ) swapElement( m, TRUE )
23       ELSE swapElement( m, FALSE )
24       RETURN
25     }
26     p ← p ÷ m
27   }
28   swapElement( 1, TRUE )
29 }
```

Time Complexity of Steinhaus-Johnson-Trotter Algorithm

Looking at the performance of the algorithm:

- The initial population of the element and position arrays and calculation of the total number of permutations can be performed in $O(N)$ time.
- The function to swap elements can be performed in $O(1)$ time.
- There are $N!$ permutations in total.
- $(N! - (N - 1)!)$ of the permutations are generated by swapping the largest (N^{th}) element; requiring a single iteration of the loop within the next permutation function.
- Of the $(N - 1)!$ remaining permutations, $((N - 1)! - (N - 2)!)$ are generated by swapping the second largest element; requiring an iteration to determine that the largest element is immobile and a second iteration to determine the second largest element is mobile and swap it.
- In general, the K^{th} element is swapped $(K! - (K - 1)!)$ times during an entire cycle through all permutations. Moving that element requires $N - K$ iterations through the loop to check that the elements N through $K + 1$ are immobile and then one final iteration to swap the K^{th} element.
- When all elements are immobile then the 1^{st} element is swapped; the loop is iterated over $(N - 1)$ times, checking that the N^{th} down to 2^{nd} elements are immobile before the 1^{st} element is swapped (once the previous elements are all immobile there is no requirement to check for mobility of the 1^{st} element as there is only one permutation where this occurs, and will reset the order).
- For an N element sequence, to cycle through all possible $(N!)$ permutations requires a total of $\sum_{k=2}^N (N - k + 1)(k! - (k - 1)!) = \sum_{k=2}^N k!$ iterations through the loop and $N!$ swaps.
 - At $N = 3$, there are a total of 6 permutations and the loop within the *nextPermutation* procedure is iterated over a total of 8 times giving an average cost of 1.3 iterations per permutation.
 - At $N = 4$, there are 24 permutations and 32 iterations, again, giving an average cost of 1.3 iterations per permutation.
 - At $N = 5$, there are 120 permutations and 152 iterations, giving an average cost of 1.26 iterations per permutation.

Generally, when $N \geq 2$, the average number of iterations over the loop within the *nextPermutation* procedure (over all $N!$ permutations) is given by $\sum_{k=2}^N \frac{k!}{N!}$ and, as N tends to infinity, the average cost tends towards 1.

The maximum cost for moving from one permutation to the next requires $(N - 1)$ iterations through the loop but this only occurs when moving the 2^{nd} element (on the $\frac{1}{2}N!$ permutation) and when resetting the permutation order by moving the 1^{st} element (on the $(N! - 1)^{\text{th}}$ permutation).

Therefore, the cost of the Steinhaus-Johnson-Trotter algorithm depends on whether the algorithm is considered in the context of: generating a single permutation from the previous permutation (1 permutation in maximum of $O(N)$ time); or generating all possible permutations ($N!$ permutations in $O(N!)$ time).

Jumping to a Permutation Index

The largest (N^{th}) indexed element starts (0^{th} permutation) at the tail of the sequence (N^{th} position) and steps towards the head until it reaches the 1^{st} position in the $(N-1)^{\text{th}}$ permutation. For the next N permutations: the largest indexed element start in the 1^{st} position (on the N^{th} permutation) and step until it reaches the tail-most (N^{th}) position on the $(2N-1)^{\text{th}}$ permutation. This pattern is repeated for blocks of $2N$ permutations. Therefore the position of the largest element can be calculated from the permutation index (C) with the given pseudo code:

```

1 IF ( modulo( C, 2 × N ) < N ) position ← N - modulo( C, N )
2 ELSE position ← modulo( C, N ) + 1
3 elementPosition( N ) ← position
4 elementIndex( position ) ← N

```

Considering the second largest indexed element, while ignoring the position of the largest indexed element, then on the 0^{th} permutation it is at the tail of the sequence ($(N-1)^{\text{th}}$ position, ignoring the largest indexed element) and on the N^{th} permutation it moves one position headwards. This is repeated every N^{th} permutation until the $(N(N-1)-1)^{\text{th}}$ permutation when it is at the head of the sequence (still ignoring the position of the largest indexed element). The pattern is again repeated moving towards the tail for the next $N(N-1)$ permutations and then cycles until all $N!$ permutations are achieved.

The second largest element can be treated in exactly the same fashion as the largest, above, with three changes:

- $(N-1)$ is substituted in place of N ;
- $\lfloor \frac{C}{N} \rfloor$ is substituted in place of C ; and
- The position of the second largest element ignores the largest element. By calculating the position of the largest element prior to the second largest element then if the largest element equals or precedes the position of the second largest element then the second largest element's position must be shifted one place towards the tail to account for this.

This can be extended iteratively to successive sub-sequences of lower indexed elements to give a general rule to calculate the position of the permuted elements:

```

30 PROCEDURE jumpToPermutation( C ) {
31   ORDER ← {}
32   FOR ( k ← N ... 1 ) {
33     i ← 1
34     m ← modulo( C, k )
35     IF ( modulo( C, 2 × k ) < k ) position ← k - m
36     ELSE position ← m + 1
37     WHILE ( i ≤ |ORDER| AND ORDER(i) < position ) {
38       position ← position + 1
39       i ← i + 1
40     }
41     IF ( i ≤ |ORDER| )
42       INSERT position INTO ORDER IMMEDIATELY PRECEDING ORDER(i)
43     ELSE
44       APPEND position TO TAIL OF ORDER
45     elementPosition( position ) ← k
46     elementIndex( k ) ← position
47     C ← (C - m) ÷ k
48   }
49 }

```

For example, the steps to generate the 23rd Steinhaus-Johnson-Trotter permutation of the sequence $\langle A, B, C, D, E \rangle$ is shown in Table 5.6.1 ($C = 23, N = 5$). The algorithm requires 5 position assignments and 7 position shifts to generate this permutation.

k	$q = \lfloor \frac{k!C}{N!} \rfloor$	$\text{mod}(q, k)$	$\text{mod}(q, 2k) < k$	Initial Position	Shifts	Permutation
5	$23 = \lfloor \frac{120 \times 23}{120} \rfloor$	3	TRUE	$5 - 3 = 2$		$\langle \ , E, \ , \ , \ \rangle$
4	$4 = \lfloor \frac{24 \times 23}{120} \rfloor$	0	FALSE	1		$\langle D, E, \ , \ , \ \rangle$
3	$1 = \lfloor \frac{6 \times 23}{120} \rfloor$	1	TRUE	$3 - 1 = 2$	D:2 $\mapsto$ 3 E:3 $\mapsto$ 4	$\langle D, E, \ , C, \ \rangle$
2	$0 = \lfloor \frac{2 \times 23}{120} \rfloor$	0	TRUE	$2 - 0 = 2$	D:2 $\mapsto$ 3 E:3 $\mapsto$ 4 C:4 $\mapsto$ 5	$\langle D, E, \ , C, B \rangle$
1	$0 = \lfloor \frac{1 \times 23}{120} \rfloor$	0	TRUE	$1 - 0 = 1$	D:1 $\mapsto$ 2 E:2 $\mapsto$ 3	$\langle D, E, A, C, B \rangle$

Table 5.6.1: 23rd Steinhaus-Johnson-Trotter permutation of the sequence $\langle A, B, C, D, E \rangle$.

The minimum number of position shifts is 0 and occurs on the 0th permutation (i.e. $\langle A, B, C, D, E \rangle$), whereas the maximum number of position shifts occurs on the 64th permutation, when the permutation is reversed (i.e. $\langle E, D, C, B, A \rangle$), and requires 10 position shifts.

In general, for an N element set, there are N initial position assignments and between 0 and $\frac{1}{2}(N^2 - N)$ position shifts¹. This means that the 0th permutation can be generated in $O(N)$ time and memory and a general permutation can be generated in $O(N^2)$ time and $O(N)$ memory.

Permutation Constraints

Constraints can be added to a permutation such that one element must always be before another element. Since the only change between one permutation and the next in the sequence is swapping a pair of neighbouring elements then it is trivial to store the Boolean validity state for each constraint and test if this swap changes the validity of the permutation against each constraint; this requires, for K constraints, $O(K)$ memory and time to go from one permutation to the next (note: this is bounded by the number of constraints and has no relation to the size of the permutation).

If a single constraint is required and it can be arranged that the lowest indexed element must be leftwards of the second lowest indexed element then: permutations 0 to $(\frac{N!}{2} - 1)$ will be valid permutations; on the $\frac{N!}{2}$ th permutation the lowest and second lowest indexed elements are swapped, invalidating the constraint, so the second half of the permutations are all invalid; until the $N!$ th transposition when the order is reset. This is a particularly useful result as the valid permutations can be cycled through until the first invalid permutation is reached and then a jump can be made back to the 0th permutation in $O(N)$ time (the same time bound as swapping the smallest and second smallest indexed elements). Therefore, in this case, the algorithm will still be bounded at $O(N)$ maximum and $O(1)$ mean time per transposition between sequential permutations.

¹ When each successively lower indexed element attempts to insert itself into the permutation at position 1 then the highest indexed element succeeds, the next highest element is shifted once to the right; the next highest element is shifted twice; and so on until the lowest indexed element is shifted right $(N - 1)$ times and is inserted into the right-most position. The total number of shifts required to accomplish this is $\sum_{k=0}^{N-1} k = \frac{1}{2}(N^2 - N)$.

Iterating Through All Permutations of a Biconnected Graph

The procedure below considers the process of iterating through all possible permutations of embedding for a biconnected graph.

Listing 5.6.1: Iterating Through All Permutations

```

1  LET  $f$                 ← NUMBER OF FLIPPABLE BLOCKS
2  LET  $p$                 ← NUMBER OF PERMUTABLE GROUPS OF SEGMENTS
3  LET FlippableBlocks ← ARRAY OF ALL FLIPPABLE BLOCKS
4  LET PermutableGroups ← ARRAY OF ALL permutable GROUPS OF SEGMENTS
5  LET FlipPermutation ← 0
6  LET TotalFlips      ←  $2^f$ 

8  PROCEDURE nextEmbedding() {
9    FlipPermutation ← FlipPermutation + 1
10   IF ( FlipPermutation ≥ TotalFlips ) {
11     FlipPermutation ← 0
12     LET  $i$  ← 0
13     WHILE (  $0 \leq i < p$  ) {
14       IF ( PermutableGroups[ $i$ ] HAS MORE PERMUTATIONS ) {
15         PermutableGroups[ $i$ ].nextPermutation()
16         IF ( PermutableGroups[ $i$ ] IS VALID PERMUTATION ) {
17            $i$  ←  $i + 1$ 
18         } ELSE {
19           PermutableGroups[ $i$ ].jumpToPermutation( 0 )
20            $i$  ←  $i + 1$ 
21         }
22       } ELSE {
23         PermutableGroups[ $i$ ].jumpToPermutation( 0 )
24          $i$  ←  $i + 1$ 
25       }
26     }
27     IF (  $i = p$  )
28       RETURN ALL_EMBEDDINGS_FOUND
29   }
30   LET  $k = 1$ ;
31   FOR (  $b = 0 \dots f-1$  ) {
32     IF ( BITAND( FlipPermutation,  $k$  ) > 0 ) {
33       FlippableBlock[ $b$ ] IS FLIPPED
34     } ELSE {
35       FlippableBlock[ $b$ ] IS NOT FLIPPED
36     }
37      $k$  ←  $k \times 2$ 
38   }
39   RETURN NEXT_EMBEDDING_FOUND
40 }
```

Considering the time bound of this procedure then, at worst, a single permutation of p elements can require a maximum time bound of $O(p)$ to get to the next permutation. Therefore with permutable graphs with $\{p_1, p_2, \dots, p_n\}$ elements the maximum time bound is of the order of $\sum_{i=1}^n p_i$. Since each permutable segment is contained in a single permutation then the upper time bound is $O(\mathcal{S})$ (where $\mathcal{S}$ is the set of segments).

Similarly, if there are f flippable blocks then the bits representing the flipped status can be set in constant time (since they are being represented by a numerical value); however to translate this back to the status of each block requires each individual bit to be queried so needing an $O(f)$ upper time bound. If each block was flippable and every non static segment was in a different block then to update the status of every block would require an $O(\mathcal{S})$ upper time bound.

Therefore, the maximum time bound to set the permutation order and flips is $O(\mathcal{S})$.

5.7 Edge Order Generation Algorithm

A cyclic edge order can be defined such that each vertex has a parent and a stem edge incident (the first inbound and outbound edges, respectively) and these edges are contained in the same segment as that vertex. Each given edge forms an element of two doubly linked cycles such that for both incident vertices there is an edge anti-clockwise and an edge clockwise from that given edge – initially, each edge can be set that it is clockwise and anti-clockwise from itself and when it, or subsequent edges, are added to the cyclic order about an incident vertex then those cyclicly adjacent edges can be updated.

Definition 4.4.3 (page 69) states how to generate a cyclic direction for each segment of a biconnected component. This translates into the algorithm below (taking into account that the containing blocks may have been flipped).

Listing 5.7.1: Setting Segment's Cyclic Direction

```

1 rootVertex.segment HAS CLOCKWISE CYCLIC DIRECTION
2 FOR ( b ∈ BlockList ) {
3   FOR ( s ∈ b.allInside )
4     IF ( b IS FLIPPED ) {
5       s IS DIRECTED IN THE OPPOSITE CYCLIC DIRECTION TO s.parent
6     } ELSE {
7       s IS DIRECTED IN THE SAME CYCLIC DIRECTION TO s.parent
8     }
9   FOR ( s ∈ b.allOutside )
10    IF ( b IS FLIPPED ) {
11      s IS DIRECTED IN THE SAME CYCLIC DIRECTION TO s.parent
12    } ELSE {
13      s IS DIRECTED IN THE OPPOSITE CYCLIC DIRECTION TO s.parent
14    }
15 }
```

For B blocks and S segments, this can be achieved in $O(S + B)$ upper time bound. However, since $B \leq S$ this can be simplified to $O(S)$ upper time bound.

Expanding this to generate a cyclic edge order then definitions 4.4.4, 4.4.5 and 4.4.6 describe how to attach the head and tail vertices within an existing cyclic edge order. The algorithm below follows from those definitions.

Listing 5.7.2: Generating the Edge Order for a Biconnected Graph

```

1 FOR ( EACH EDGE  $(u \rightarrow v)$  ∈ ROOT segment )
2   v.parent ←  $(u \rightarrow v)$ 
3 FOR ( EACH EDGE  $(u \rightarrow v)$  ∈ ROOT segment ) {
4   u.stem ←  $(u \rightarrow v)$ 
5   u.stem IS adjacent BOTH ANTI-CLOCKWISE AND CLOCKWISE FROM u.parent ABOUT u
6   u.STACK ← EMPTY STACK
7 }
9 previous ← ROOT segment
11 FOR ( s ∈ SegmentList \ {ROOT segment} ) {
12   WHILE ( previous ≠ s.parent ) {
13     POP FROM previous.getTailVertex().STACK
14     previous ← previous.parent
15   }
16   FOR ( EACH TREE EDGE  $(u \xrightarrow{T} v)$  ∈ s )
17     v.parent ←  $(u \xrightarrow{T} v)$ 
18   FOR ( EACH NON HEAD EDGE  $(v \rightarrow w)$  ∈ s ) {
19     v.stem ←  $(v \rightarrow w)$ 
```



```

20      v.stem IS adjacent BOTH ANTI-CLOCKWISE AND CLOCKWISE FROM v.parent ABOUT v
21      v.STACK ← EMPTY STACK
22  }

24  LET h ← s.headVertex()
25  LET t ← s.tailVertex()
26  LET e ← s.tailEdge()
27  LET d ← cyclic direction OF s

29  add s.headEdge{} TO cycle ABOUT h adjacent TO h.parent IN opposite direction
  TO d

31  IF ( t.STACK IS EMPTY ) {
32      add e TO cycle ABOUT t adjacent TO t.stem IN direction d
33  ELSE {
34      peek at ⟨a, e, c⟩ on t.STACK
35      IF ( d IS CLOCKWISE )
36          add e TO cycle ABOUT t CLOCKWISE adjacent TO a
37      ELSE
38          add e TO cycle ABOUT t ANTI-CLOCKWISE adjacent TO c
39  }

41  LET a ← EDGE ANTI-CLOCKWISE ABOUT t FROM e
42  LET c ← EDGE CLOCKWISE ABOUT t FROM e
43  PUSH ⟨a, e, c⟩ ONTO t.STACK
44  }

```

Considering listing 5.7.2:

Lines 1-7 consider the root segment and iterates over the segment's edges attach the parent (inbound) and stem (outbound) edges to each vertex and sets their initial cyclic order and sets an empty stack to contain structures needed for later embeddings. The time complexity of this operation is proportional to the number of edges in the root segment.

Lines 12-15 handles backtracking when the algorithm has embedded all descendants of a given segment and the vertex stacks can have the triplet of cyclically adjacent edges relative to those descendants' tail edges removed as they are no longer relevant to the embedding. Each segment can only be added to, and hence removed from, the stack once so when iterating over all segments this check will be performed once per non-root segment and at most each segment can be popped from the stack once so in total this is $O(S)$.

Lines 16-22 attach the parent and stem edges to each vertex about the non-root segments and set an empty stack to contain structures needed for later embeddings. Since each edge is contained in exactly one segment and each segment is iterated over exactly once then this and lines 1-7 are performed in $O(\mathcal{E})$ time.

Line 29 attaches the segments head edge to the head vertex (as per definition 4.4.4); this can be performed in constant time (since this is the addition of an element to a doubly-linked list).

Line 32 attaches the segments tail edge to the tail vertex in the case where no other ancestors (except the containing segment) are attached (as per definition 4.4.5) and can be performed in constant time.

Lines 34-38 attach the segments tail edge to the tail vertex in the case where other ancestors are attached (as per definition 4.4.6) and can be performed in constant time.

Therefore, lines 29-38 can be performed in constant time so these operations can be performed for all segments in $O(S)$ time.

Examining the time complexity of the procedure then each segment is considered as is each edge and the total time complexity of the procedure is $O(\mathcal{E} + \mathcal{S})$.

5.8 Planarity Testing Conclusions

This chapter has shown how to:

- Generate a Trémaux Tree in $O(\mathcal{V} + \mathcal{E})$ upper time bound;
- Partition the graph into segments in $O(\mathcal{V} + \mathcal{E})$ upper time bound;
- Test the graph for planarity in $O(\mathcal{S})$ (where $\mathcal{S}$ is the segments of the graph) and whilst performing this test identify *flippable*, *permutable*, *rotatable* and *reversible* segments;
- Generate a permutation of the graph in a $O(\mathcal{S})$ upper time bound; and
- Generate a cyclic edge order for that embedding in $O(\mathcal{E} + \mathcal{S})$ upper time bound.

Since, for connected graphs, both the number of segments and the number of vertices is less than or equal to one plus the number of edges then this can be simplified to conclude that all phases of the algorithm have an $O(\mathcal{E})$ upper time bound and that the execution of the algorithm is linear with respect to the number of edges of the graph.

Chapter 6

Benchmarking Methodology

This chapter is a discussion of the process used to benchmark the performance of the algorithm. It will discuss:

- What is software performance benchmarking;
- Factors affecting the robustness of a software benchmark;
- Mitigating against benchmarking factors; and
- Analysis techniques to reduce the impact of unmitigated factors.

6.1 What is software performance benchmarking?

Software benchmarking is the process of executing a given piece of software to test it against a specified metric to give an indication of the software's performance [48, 86]. "Performance" could be measured in terms of:

- Execution speed, where a smaller execution time (for the same effect) indicates higher performance;
- Throughput, where a greater amount of processing in a given unit of measurement indicates higher performance. This is typically measured against time (i.e. frames per second in 3D graphics or transactions per second for database applications) but can be applied to other units such as cost, when looking at performance compared to total-cost-of-ownership, or Watts, when measuring the performance of portable devices where battery life is important; or
- Correctness, where the benchmark is designed to validate the software against a set of predetermined criteria.

In a perfect world the software being benchmarked would execute in a constant time every repetition and the process of benchmarking software would be a matter of simple comparison, unfortunately this does not happen as various factors affect the result. Therefore a significant effort is required to make sure that these factors are eliminated or minimised during the execution of the benchmark or in the analysis. When comparing benchmarks, they need to be comparable, repeatable and robust.

- Benchmarks are comparable when they are executed under identical conditions and variation in the results can be attributed to changes in the input conditions to the benchmark rather than to external factors. This means that when two benchmarks are compared they should have been executed using the same source code; compiled in the same way; executed using identical settings; and run in comparable environments (i.e. on comparable hardware and operating system).
- Repeatable benchmarks are ones where the results can, within an acceptable margin of error to account for external factors, be reproduced given comparable execution settings and environments. Typically, a single instance of a benchmark is not enough to quantify performance and a series of comparable benchmarks should be used to give an estimate of the “average” performance so that extreme values can be detected as outliers and mitigated against.
- Robust benchmarks ensure that the benchmarked quantity that which is measured and there are no other factors obscuring or distorting the measurement. This means that during the execution of the benchmark the external factors influencing the results are minimised and, during analysis, robust statistical measures are used to minimise the effect of outliers resulting from the external factors that cannot be eliminated.

The remainder of this chapter will look at the benchmarking process and factors affecting the benchmarking of a planarity testing algorithm written in Java to test the execution speed against the hypothesis that the execution time increases linearly with graph size.

6.2 Factors affecting the Robustness of a Software Benchmark

There are three broad areas where effects can impact on a benchmark; these are: software, hardware and environmental factors.

Software factors are the effects of software other than the benchmarked application on the benchmark result. Typically these can be categorised into sub-groups relating to: the operating system (and can be further categorised into critical and non-critical services); processes and services the benchmark is reliant upon (i.e. a JVM or integrated development environment); and other applications. The impact of these factors is likely to be significant as other software will take processor time away from the benchmarked software and one interruption can result in a cascade effect where a later interruption may occur during the benchmark time frame because that time frame was extended due to an earlier interruption.

Typically, benchmarking software or programming languages will have options (i.e. command line options on the JVM) to log interruptions (such as Just-In-Time compilation or Garbage Collection) or these can be built into the benchmarking software. However, when considering software that is more removed from the testing environment, it can be more difficult to monitor (and when there is the capability to monitor, it then the issue is synchronisation with the benchmarking process) and these factors are likely to have a significant impact on the benchmark and need to be mitigated against. Often the simplest mitigation is to stop as many programs and services from running as possible but, to be able to compare benchmarks, a consistent setup should be maintained during subsequent benchmarks.

Considering the impact hardware has on benchmarking the execution time of an algorithm, there are very little in the way of hardware requirements. These include: a minimal user input to initiate the benchmarking process (but after that the process can be automated and

there is no further user input); an output requirement to maintain a log of the benchmarked performance; and the most basic hardware requirement is the processing capability to perform the benchmark. Therefore the number of hardware factors can be immediately reduced to the smallest sub-set able to meet these capabilities and eliminate any additional factors that may arise from additional, unnecessary hardware.

Of the required hardware capabilities:

- User input is easily managed as it is only required to initiate the benchmark and thereafter the user can be disconnected to eliminate any variables this may introduce to the process.
- Processing capability is a necessity and is, naively, assumed that for repeated execution of a benchmark the typical load on the hardware will be consistent; it is when additional software factors (i.e. the operating system) are involved that the performance obtained from the hardware is reduced and these are the factors that need mitigating by terminating as many software applications and services as possible before benchmarking.
- Output is the most significant overhead, when considering hardware factors, as processing can, with sufficient resources, be performed in memory whereas recording the benchmarking result requires writing to disk and incurs a significant relative delay. However, logging results can, and should, be performed in the time between benchmarks and so mitigation for delays due to logging relies on the test suite accurately partitioning the results into time spent benchmarking a time spent on other tasks so as not to have erroneous results.

Environmental factors are considered to be external to the hardware and software and can include temperature, humidity, power supply, or other similar effects; they are either small gradual changes, such as temperature variations, that are difficult to quantify the effect on performance; or are low-probability dramatic changes, such as power failure, that are typically terminal for the benchmarking process. Therefore, while their impact is noted for completeness, these factors are generally accepted to not be significant to the overall validity of a benchmark.

In conclusion: environmental factors are believed to be either of negligible impact or have very low probability of occurrence and will be discounted. The major hardware factors can be mitigated for by using the minimum necessary hardware setup; removing the user from the system (as much as is practical); and making sure the logging of results is performed outside of the benchmarking period. The impact of software can be reduced, as with hardware, by using a minimal setup and when interruptions occur, they should either be monitored during benchmarking or, where appropriate, treated as outlying results and removed from the datasets. A more detailed discussion of specific software factors is included below.

6.3 Mitigating for Software Factors during Benchmarking

This section deals with more specific software factors relating to benchmarking and, in particular, those relating to the Java Programming Environment. These include:

- Clock Resolution;
- Code Warm-up (Class Loading and Just-in-Time compilation); and
- Garbage Collection

6.3.1 Clock Resolution

Timings in Java can be obtained using several methods [6, 71]:

1. `System.currentTimeMillis()` returns the current time in milliseconds (ms), since 1st January 1970, but may have a resolution of tens of milliseconds depending on the operating system; this low resolution is a problem if the execution time approaches this limit. If the system clock is changed during a benchmark (due to changes in daylight savings time or Network Time Protocol synchronisation) this will result in erroneous reporting.
2. `System.nanoTime()` returns the elapsed time from an arbitrary static time frame and is not related to the system clock or a wall clock. It is reported to nanosecond (ns) precision but is not necessarily nanosecond accuracy.
3. `java.lang.management.ThreadMXBean.getCurrentCPUTime()` returns a nanosecond precision measurement based on CPU time for the thread; this is typically more resistant to interruptions than `System.nanoTime()` but the time for some operations, such as I/O, is platform dependent as to whether it is allocated to the program thread or a system thread. It also relies on native methods to access this information and so is not supported on all JVMs or platforms (and on the platform used for testing this method was indicated as being supported but always returned a time of zero nanoseconds). The other main issue with this method is that it is based on thread CPU time and if the process spawns multiple worker threads then the returned CPU time for the main thread may not accurately reflect the actual time spent on the task.

The preferred method is to use `System.nanoTime()` as it is more robust than `getCurrentCPUTime()` and, typically, has a better resolution than `System.currentTimeMillis()`. The resolution of `System.nanoTime()` was tested using the sample code, shown in Appendix C.1, and the results are summarised in Table 6.3.1 (shown in full in Appendix C.2); this shows the best resolution appears to be of the order of 1 microsecond (μ s), with a mean of 1.3666ms, but even simple code (with sequential calls to record the time) can be interrupted by external factors and have delays of the order of 10ms.

Clock Resolution (ns)	Frequency (out of 10,000,000 repetitions)	Cumulative Frequency (%)
977	555,426	5.55426
978	1,940,982	24.96408
1466	2,493,535	49.89943
1467	5,000,942	99.90885
1955	1,813	99.92698
1956	2,281	99.94979
2444	1,277	99.96256
2445	1,001	99.97257
...	...	...
9778	39	99.97736
10266	15	99.97751
...	...	...
99734	1	99.99499
100222	3	99.99502
...	...	...
6486579	1	99.99999
17440136	1	100.00000

Table 6.3.1: Summary of Clock Resolution Times

The execution time results showed a rough pattern: there are pairs of times 1ns apart; and the increment between successive pairs is, typically, 489ns or a multiple thereof. An assumption can be made that the results 1ns apart are due to rounding to the nearest nanosecond and the increment is related to the duration of one clock cycle but this is only supposition. It does highlight that there is a degree of granularity to the results and accuracy below approximately 500ns does not appear to be achievable.

The results also show it is possible for significant interruptions to occur between sequential calls to the timer and that: 0.0225% of the results showed a clock resolution above $10\mu\text{s}$; 0.005% had a resolution above $100\mu\text{s}$; and the minimum recorded resolution was of the order of tens of milliseconds (over 10,000 times greater than the maximum resolution). This shows the importance of taking repeated measurements and the impact that extreme outliers could have on the results as the full dataset has mean of 1367ns and standard deviation of 6199ns but if the results are cut off at 3 S.D. above the mean then 99.95% of the results remain and the mean is approximately the same at 1345ns but the standard deviation is significantly reduced to 224ns and is a closer representation of the spread of the data.

The conclusion from this test is that the maximum resolution provided by `System.nanoTime()` is approximately $1\mu\text{s}$ (achieved in approximately 30% of the results) but approximately 99.95% of the test show a resolution of $2\mu\text{s}$ or less and, therefore, the clock resolution should be considered accurate to the order of the nearest $10\mu\text{s}$.

6.3.2 Code Warm-Up

Code warm-up encapsulates two problems: class loading/unloading and just-in-time compilation; both of these effects affect the results during the time it takes the JVM to “warm-up”.

Class Loading

Class loading [86, chapter 6] is the process by which the Java Virtual Machine (JVM) loads the classes ready for first execution and typically involves disk I/O, parsing and verification; all of which greatly inflate the execution time of the first cycle through the program. In addition, the JVM can also decide to unload classes that have become garbage.

There are two methods of detecting class loading and unloading:

- The `java.lang.management.ClassLoadingMXBean` class has methods `getTotalLoadedClassCount()` and `getUnloadedClassCount()` that can be used to poll the JVM before and after benchmarking and an increase in either count indicates that class loading or unloading has occurred; or
- The command line options `-XX:+TraceClassLoading` and `-XX:+TraceClassUnloading` will display on the standard output (if a console has been allocated) when a class is loaded or unloaded.

Neither of these methods will print any timing data to show the performance impact that class loading has on the execution time.

Testing the algorithm to determine when the classes are loaded shows that the majority of the classes are loaded prior to benchmarking and, as expected, the remaining classes are loaded during the first execution cycle. Therefore the first result in the dataset should be considered suspect as it is likely to have been influenced by the effects of class loading. During execution, no instances of class unloading were detected so it is believed not to be a factor in the algorithm’s performance but is mentioned for completeness.

Just-in-Time Compilation

Just-in-Time (JIT) compilation is a feature of modern mixed-mode JVMs; the code is initially allowed to run in a purely interpreted mode while it profiles the execution and then after a number of cycles a JIT compilation will be performed on a given code block, where a class can be formed of many nested code blocks, and from then onwards the compiled code will be executed. This compiled code is used until the JVM detects that any of its assumptions used to compile the code are outdated, at which time de-optimisation occurs and the code block will return to running in interpreted mode and profiling will begin again; one example of this is “uncommon traps” [6] where the most likely path is compiled initially and atypical branches, such as exception paths, are left as interpreted code blocks but if detected may trigger recompilation of the code.

There are two methods of detecting JIT compilation:

1. The command line option `-XX:+PrintCompilation` will display an output to the standard output stream (assuming a console has been allocated) when a compilation occurs detailing (for the JVM used in testing): the sequential index given to the compilation; an identifier for the code block; and the number of bytes. It does not display the time spent performing compilation.
2. `java.lang.management.CompilationMXBean` has a `getCompilationTime()` method that returns the total time (in milliseconds) spent by the JVM in compiling code and by polling this at the start and end of the benchmark the time spent compiling (to the nearest millisecond) can be obtained.

The following two snippets of the algorithm’s output highlight problems with using these methods. In both cases the output has been modified to display as “execution time” minus “compilation time” (calculated using the `CompilationMXBean` interface and polling before and after the benchmark) equals “adjusted time” (all units are in nanoseconds). The code was executed with the command line option enabled so that both methods can be compared.

Listing 6.3.1 shows execution of the algorithm that is interrupted during the generation of edges sides as the JVM compiles the `java.lang.String::indexOf` code snippet (line 6); However, the compilation time shows that 0 milliseconds were spent compiling code (line 7) as the total compilation time is likely to be less than 0.5 milliseconds and therefore rounded down to give a result of 0; this means that the millisecond resolution can negatively affect the results.

Listing 6.3.1: Code Compilation – Rounding Errors

```

1 Root: 1
2 Timing Started
3 DFS Completed: 6328667 - 0 = 6328667
4 Segment Generation Completed: 3924312 - 0 = 3924312
5 Segment Embedding Completed: 4158000 - 0 = 4158000
6 1 java.lang.String::indexOf (166 bytes)
7 Edge Sides Generated: 4445956 - 0 = 4445956

```

Listing 6.3.2 shows two features: firstly that the compilation highlighted on lines 10-11 took 12 milliseconds to execute, however the code is recorded as having taken approximately 1.5 milliseconds to execute and therefore the adjusted time is negative; and secondly, line 13 shows that the segment embedding was interrupted for 4 milliseconds for compilation (again giving a negative adjusted time) but there was no compilation displayed, immediately prior to this line, using the command line option whereas three compilations (lines 14-16) interrupt the phase succeeding it (line 17).

Listing 6.3.2: Code Compilation – Sequencing Errors

```

8 Root: 1
9 Timing Started
10 8      mgtaylor.datastructures.UnmodifiableLinkedList::addTail (141 bytes)
11 DFS Completed: 1514089 - 12000000 = -10485911
12 Segment Generation Completed: 1156222 - 0 = 1156222
13 Segment Embedding Completed: 2064089 - 4000000 = -1935911
14 9      mgtaylor.datastructures.UnmodifiableLinkedList::getTail (17 bytes)
15 11     mgtaylor.datastructures.UnmodifiableLinkedList$1::next (30 bytes)
16 10     mgtaylor.graph.Edge::isDFSStem (18 bytes)
17 Edge Sides Generated: 2993956 - 1000000 = 1993956

```

These results show that:

- The millisecond resolution provided when polling the JVM is not always fine enough to detect when a JIT compilation has occurred;
- There is a discrepancy between the duration of JIT compilation (milliseconds) and the more accurate benchmark times (nanoseconds) and can result in a negative adjusted time;
- The command line option does not give any feedback on the time spent on compilation and so is of limited use in benchmarking; and
- The command line option only appears to identify when compilation of a code fragment has completed and is not a reliable indicator of how the compilation time is spread.

Effects of Code Warm-Up

Figure 6.3.1 shows a scatter plot for the execution times of the algorithm on a 1250 vertex triangular prism graph (described in section 7.3.1) where benchmarking 1000 repetitions of the algorithm has occurred immediately after initialising the JVM and demonstrates several features:

- The top plot shows that the first result is of a different order of magnitude from the majority of the results and is indicative of the effect of class loading;
- The results affected by garbage collection are spaced at approximately regular intervals. The bottom scatter plot (showing the results up to the 98th percentile) shows a garbage collection occurred every 55-65 repetitions – equivalent to a garbage collection approximately every 3 tenths of a millisecond;
- Over the period up to the 7th garbage collection (389th repetition), there are steps in the scatter plot reflecting a gradual improvement in the execution representative of the effects of JIT compilation; and
- After the 7th garbage collection, the plot stabilises to have to a mean execution time of 10.82 ms (with a standard deviation of 0.65 ms) and the effects of code warm-up appear to have disappeared.

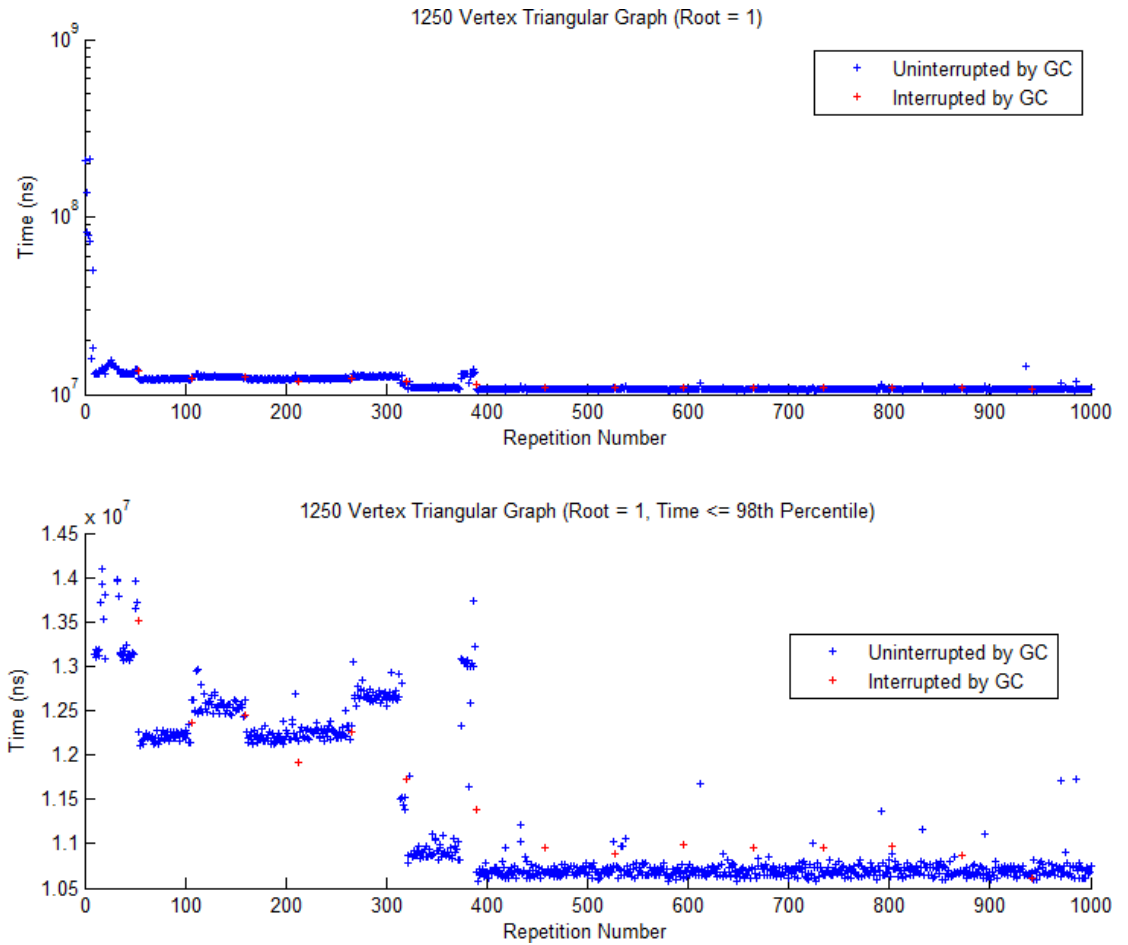


Figure 6.3.1: Code Warm-Up for 1250 Vertex Triangular Graph (Root = 1)

Immediately after the algorithm was run for another 1000 repetitions, this time using the 2nd vertex as the root of the graph; comparing the results shown for the 1st (Figure 6.3.1) and 2nd (Figure 6.3.2) vertices as roots shows that once the JVM has performed any JIT compilations and is stable that, with the exception of a few outliers and the results affected by garbage collection, the results are approximately constant (for 2nd vertex as the root, mean is 11.36ms and standard deviation is 0.18ms).

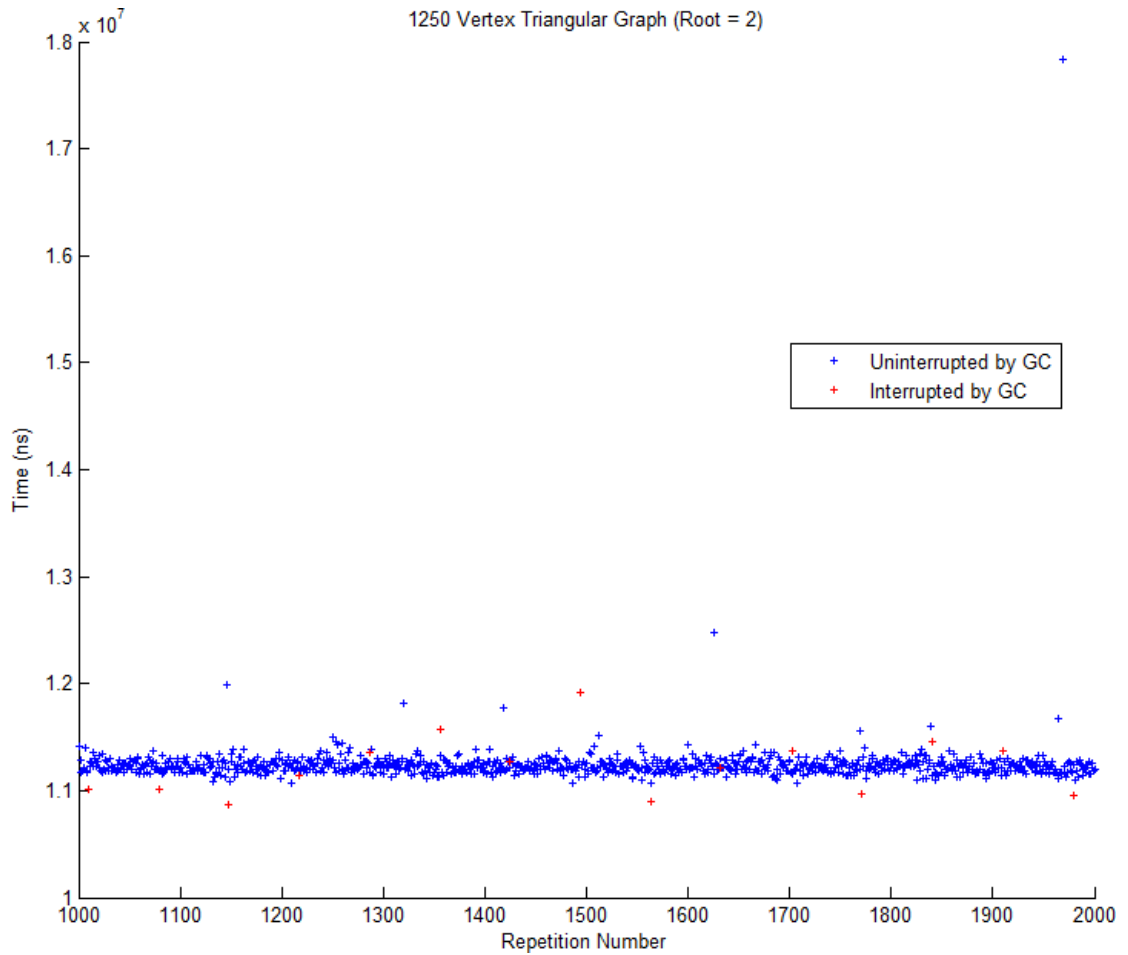


Figure 6.3.2: Code Warm-Up for 1250 Vertex Triangular Graph (Root = 2)

Mitigating against Code Warm-Up

Without a deeper understanding of the workings of a JVM, it is difficult to make sense of the nuances of how compilation occurs and is reported; however, the conclusion can be drawn that both class loading and JIT compilation present a problem to performing benchmarking tests and it is difficult, in both cases, to accurately quantify their effect.

There are two ways to mitigate against Code Warm-Up:

- Run through a number of iterations through the algorithm before recording any data ensuring that the majority of the classes are loaded and JIT compilation has occurred. The Java Hotspot Server Virtual machine profiles a given code block for 10,000 iterations before performing compilation [6] therefore a number of repetitions in excess of this should insure that the code has been “warmed-up” and will run in a steady state. This will not cover instances where rarely used code segments are accessed and in these cases it may take longer to reach the point at which JIT compilation occurs and will have to be mitigated against in another way; such as
- Removing or reducing the impact of the upper percentiles of data in the dataset. Code warm-up increases the execution time and means that data points affected by this will fall into the upper percentiles of the data set and can be treated as outliers and handled either via pruning the data set (ensuring that removed results are outliers and not part of a distribution with a heavy tail) or using analysis techniques that can robustly handle outliers, to minimise the distortion these create.

6.3.3 Garbage Collection

Of the interruptions that occur during execution, garbage collection (GC) is the most frequent and regularly reoccurring issue; this means that dealing with the effects of garbage collection is one of the most pressing problems with benchmarking.

Java provides several methods of analysing the impact garbage collection has on the performance, including:

- The `java.lang.management.GarbageCollectorMXBean` interface provides two methods to query the JVM regarding the total number of garbage collections that have occurred and the total collection time (in milliseconds). This interface allows the JVM to be polled before and after a benchmarking loop to calculate the increase in time spent on garbage collection.
- The command line option `-XX:PrintGCDetails` adds verbose output to the standard output channel (if the console is enabled).

There are 4 issues that need to be taken into account when analysing the impact of garbage collection:

- Timing resolution for garbage collection reporting methods;
- False positive or negative reporting of garbage collection instances;
- Similarity/dissimilarity of data sets affected and unaffected by garbage collection; and
- The frequency garbage collection as graph size increases.

Garbage Collection Timing Resolution

Considering the timing resolution, the `GarbageCollectorMXBean` interface is specified to have millisecond granularity (about 1000 times worse than the clock resolution, see section 6.3.1) and therefore rounding errors in the reported garbage collection time will be significant for results that are of a similar or smaller order of magnitude; as the typical execution time of a 1000 vertex Triangular Prism graph is less than 9 milliseconds, this is a significant issue. Compared to this, the command line option provides timing details with a 100 nanosecond precision; the accuracy of these results is not commented on in the API documentation but can be assumed to have an accuracy that is at best the same as the clock resolution (approximately 1-2microseconds). Given the resolutions provided by the two methods, it is preferable to use the command line option for the greater reported accuracy.

Using the command line option requires that the start and end of any benchmarking period is output to the standard output channel and the console output must be synchronised with this (see section 6.3.4 for more details of synchronisation issues) so that garbage collection reporting is correctly ordered within the log file. The information collected from the `GarbageCollectorMXBean` interface can be polled directly from the JVM and integrated into the stream used by the benchmarking process to log results and so does not require synchronisation with the console; this may result in a reduced error rate due to any synchronisation issues but this is of lower impact than the issues created from the lower timing accuracy.

False Positive Reporting of Garbage Collections

False positive reporting of garbage collection occurs when the JVM reports a garbage collection and it is falsely determined to have occurred during a period of benchmarking when it did not and can result in execution times lower than expected (or even negative, similar to the log snippets shown in section 6.3.2). The pseudo-code snippet below (Listing 6.3.3) gives the typical structure for performing a benchmark and identifies two areas where, if interrupted by garbage collection during this period of execution, a false positive garbage collection will be reported. These areas are impossible to eliminate and mitigation for them has to take the form of ensuring that there are as few operations between logging that the benchmarking process has started (ended) and recording the start (end) time for the benchmark.

Listing 6.3.3: Garbage Collection – False Positive Results

```
1 log( "Start Benchmark" );
3 // False positive area #1
5 startTime = getTime();
6 performBenchmark();
7 endTime = getTime();
9 // False positive area #2
11 log( "Benchmark Finished" );
12 elapsedTime = endTime - startTime;
13 log( "Benchmark Time: " + elapsedTime );
```

Examining the 1000 vertex triangular prism graph dataset used in the next chapter, false positive results were easily identifiable as the garbage collection time was greater than the execution time and so the adjusted time (execution time minus garbage collection time) was negative. Looking at the error rate there was less than 10 out of 1,000,000 results where a false positive detection of garbage collection was made and therefore the overall impact is likely to be negligible.

While the garbage collection time is greater than the total execution time this will result in a negative adjusted execution time, which is nonsensical and simple to treat as an outlier which can be removed from the dataset. When the total execution time is greater than the garbage collection time then determining whether the result is a false positive detection is more difficult but can be achieved by looking for garbage collection events where the total adjusted execution time is significantly lower than the corresponding minimum execution time for the data not interrupted by garbage collections; while this is not impossible to identify it can be subjective when setting the threshold for identifying false positives. A more appropriate method is to use robust analysis techniques (discussed in section 6.4) that account for outlying results when analysing the results.

False Negative Reporting of Garbage Collection

False negative results occur when garbage collection occurs during the benchmarking process but it is not reported (or is reported outside of the logged time frame where the benchmark occurred). This can be due to: the order in which logging and timing occurs; synchronisation issues between logging the benchmark results and the JVM output that reports on Garbage Collections (see section 6.3.4); or due to garbage collection executing concurrently with the benchmark but not being reported until completion of the GC, which may be outside of the benchmarking timeframe or even during a different benchmark.

Listing 6.3.4 (where the order of starting/ending logging and timing is reversed from listing 6.3.3) gives an example of when false negatives can occur if GC occurs whilst timing but outside of logging that timing is occurring. Given that false positive results, typically, result in negative adjusted execution times (as discussed above), and are easily identifiable, whilst false negative results have no distinguishing features from outliers generated by other unidentified factors then it is better to use the structure of listing 6.3.3 to log results as it will reduce the impact of outliers.

Listing 6.3.4: Garbage Collection – False Negative Results

```
1 startTime = getTime();
3 // False negative area #1
5 log( "Start Benchmark" );
6 performBenchmark();
7 log( "Benchmark Finished" );
9 // False negative area #2
11 endTime = getTime();
12 elapsedTime = endTime - startTime;
13 log( "Benchmark Time: " + elapsedTime );
```

Considering synchronisation: logging occurs via the standard output stream whilst reporting GC interruptions occurs through the Console interface; having these two reporting methods unsynchronised can introduce errors in the reporting process (either false positive or false negative). Mitigating for synchronisation issues is discussed in more depth in section 6.3.4 but as long as the console and standard output can remain synchronised then instances of GC are reported inline with the logging and this is considered to be of low impact.

Considering concurrent GC: Normally, garbage collection pauses execution of the program and restarts it after the process has completed so there are no concurrency issues. Concurrent garbage collection is a side effect of activating the JVM with the non-standard option (`-Xincgc`) and, therefore, it is considered to be of low impact as it can be managed during the setup of the empirical testing.

If a false negative result does occur then it results in an increased execution time and can introduce an outlier within the data set; this can be mitigated for using robust analysis techniques (discussed in section 6.4) that are resistant to the presence of outliers.

Similarity of Data Affected and Unaffected by Garbage Collection

A benchmark consists of a set of repeated executions of the algorithm for a given, static, set of input conditions (graph type [see Section 7.3], size, root vertex) and the output of each repetition within the benchmark includes a report on the duration of algorithm’s execution; this allows the “average” duration of the algorithm’s execution to be calculated and compared with other benchmarks. Also output is whether that execution period was reported to have been interrupted by garbage collection and the duration of that interruption, so an adjusted time can be calculated for those repetitions subtracting the duration of GC from the total duration of execution. This allows a benchmark to be partitioned into disjoint data sets affected and unaffected by garbage collection.

It is important to ensure that the results affected by garbage collection, after adjusting for the reported GC time, do not distort the unaffected data. If the data sets are dissimilar then, even after adjustment based on the reported GC duration, there are still additional factors, compared to the non-GC data set, that are influencing the location or shape of the distribution and so inclusion of the GC data may skew any calculation of an “average” execution time for that benchmark. In this case, the GC data should not be used in the analysis as it is not representative of the rest of the (non-GC) data.

If the data sets are not dissimilar then there is a lack of evidence that the GC data is influenced by additional factors and so the conclusion can be drawn that GC could be mitigated against by calculating the adjusted execution time.

Examining the similarity (or lack thereof) between the datasets affected and unaffected by garbage collection requires a measure of similarity. This can be found using, amongst others¹: the Two-Sample Kolmogorov-Smirnov (K-S) Test [70], which considers the maximum distance between the empirical cumulative distribution functions (*ECDF*) of each sample; or the Two Sample Cramér-von-Mises (CvM) Test [1, 70], which considers the sum of the squared distances between the *ECDF*s. Both are non-parametric, distribution-free tests to quantify the distance between the *ECDF* of two samples. The null hypothesis for each test is that the samples have both been drawn from the same distribution (or from distributions with similar location and shape) – rejection of the null hypothesis indicates that the two samples are drawn from distributions that are not similar in location and shape.

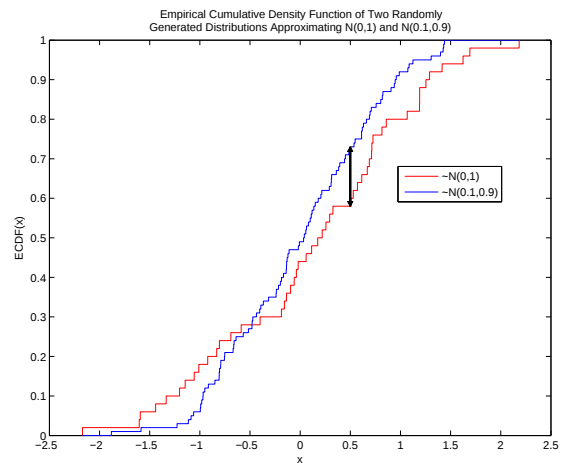


Figure 6.3.3: Example comparing *ECDF*s of samples drawn from two normal distributions.

Figure 6.3.3 shows the *ECDF* step “curves” for two samples of data, drawn randomly from the normal distributions $N(0, 1)$ and $N(0.1, 0.9)$. The region between the two step “curves” delimits the (vertical) distance between *ECDF*s at each data point, as used by the CvM test, and the vertical arrow marks the maximum (vertical) distance within this region, as used by K-S test.

¹ Other tests include: Kuiper’s test [70], a modification of the K-S test to consider the maximum deviation above and below the two *ECDF*s; Student’s *t*-test, a non-distribution-free test, used to compare two samples drawn from normal distributions with different means and equal (or non-equal for Welch’s *t*-test) variations; or Hellinger Distance [45] (and the related Bhattacharya distance) for comparing lack of similarity (distance) between two distributions.

For a given graph, a set of benchmarks (where each benchmark is a series of repeated executions of the algorithm for that graph using a single vertex as root) can be generated such that there is one benchmark for each root vertex; each benchmark can then be partitioned into the samples affected and unaffected by GC and a rejection/non-rejection of the null hypothesis, that the samples affected and unaffected by GC are drawn from distributions with similar location and shape, can be calculated, using K-S and CvM tests, for that benchmark. Table 6.3.2 lists the percentage rejection of the null hypothesis from the set of benchmarks covering all root vertices of various graph types and sizes. I.e. Table 6.3.2 shows for a 500 vertex Triangular Prism Graph, giving 500 benchmarks (one for each root vertex), that: in 99.4% (497 out of 500) of those benchmarks the K-S test rejected the null hypothesis, indicating that the samples affected and unaffected by GC are from dissimilar distributions; and in 100% (500 out of 500) of those benchmarks the CvM test rejected the null hypothesis.

$ \mathcal{V} $	Null Hypothesis (H_0) Rejected (% of $ \mathcal{V} $ Root Vertices)							
	Triangular Prism		Planar Wheel		Ordered Face Trisection		Random Face Trisection	
	K-S	CvM	K-S	CvM	K-S	CvM	K-S	CvM
250	100.000	100.000	99.200	100.000	99.200	99.600	100.000	100.000
500	99.400	100.000	100.000	100.000	97.200	97.800	100.000	100.000
750	84.267	85.200	99.333	99.467	66.533	57.600	52.400	52.800
1000	87.000	90.500	90.800	91.300	90.000	93.000	51.200	43.200
1250	93.120	92.720	91.600	92.560	98.400	99.280	50.000	52.320
1500	98.867	98.200	85.133	87.800	99.200	98.867	88.800	84.400

Table 6.3.2: Dissimilarity between Datasets Affected and Unaffected by Garbage Collection

These results show a very high rate of rejection for the null hypothesis indicating that, even after subtracting the recorded duration of the garbage collection from the benchmarked time, the data-sets affected and unaffected by garbage collection are dissimilar. The results for Random Face Trisection Graphs do show a marked reduction in the rejection rate when compared to the other results; however the rejection rate is still, typically, above 50% which suggests that the inclusion of garbage collection tainted results would distort the distribution and so should not be included in the data-set.

Frequency of Garbage Collection as Graph Size Increases

Garbage collection (GC) is the most obvious external factor influencing the results and, as discussed previously in this chapter, is relatively simple to identify and can be mitigated against by removing the data tainted by GC from the analysis. In doing so, this reduces the data set and increases the chance that outliers will have a greater impact on the results. As such it is important to understand when a saturation point will be reached and 100% of the results are likely to be tainted by GC.

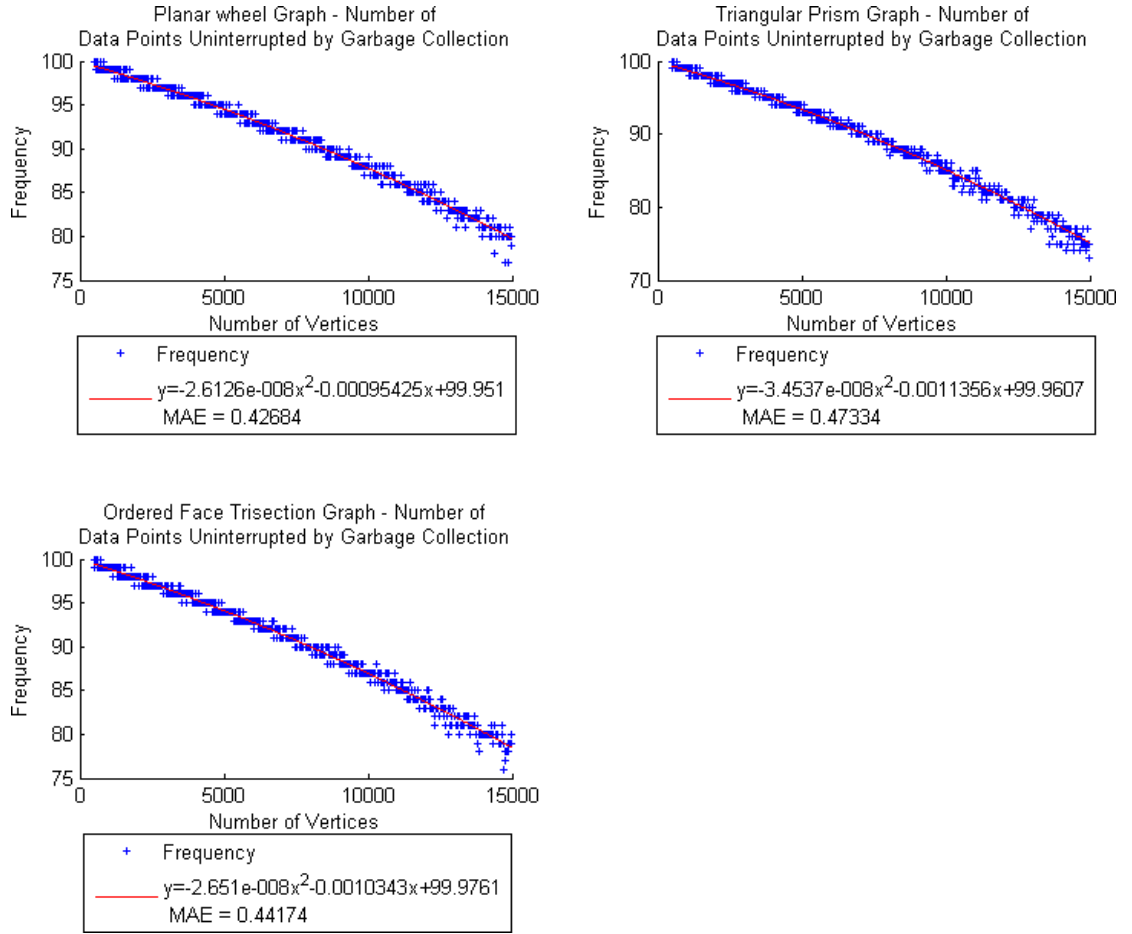


Figure 6.3.4: Number of Data Points (Out of 100) Unaffected By GC for Different Graph Types (Root = 1st Vertex)

Figure 6.3.4 shows the number of data points untainted by GC (out of a sample size of 100 for each graph size) and exhibits an, approximate, trend that $F = -3 \times 10^{-8}V^2 - 1 \times 10^{-3}V + 100$ (where F is the percentage of the data points that are untainted by GC and V is the graph size) and the reverse: $V = \frac{\sqrt{13-0.12F}-1}{6 \times 10^{-5}}$. Given this prediction then 100% of the results will be tainted by GC for graphs of approximately 43400 vertices.

Given that as graph size increases then so does the execution time of the algorithm and the proportion of results tainted by GC also rises then maintaining a reasonable sample size of untainted data becomes increasingly (and prohibitively) time consuming. The conclusion that can be drawn from this data is that for a large proportion (75%) of the data set to be untainted by GC, then the graph size needs to be less than 16500 vertices.

Mitigating against Garbage Collection

Garbage Collection can be mitigated against by:

- Recording when garbage collections occur and ensuring that this is synchronised with the algorithm's output.
- Remove any data points tainted by garbage collection from the data set (this deals with the issues of lack of similarity between the distributions for data tainted and untainted by GC and of false positive recordings of garbage collection).
- Limiting the number of vertices in a graph; this is for practical reasons and prevents large proportions (or the entirety) of the data set being affected by garbage collection and requiring that a disproportionately large sample size be generated to obtain a small sample of data that is untainted by garbage collection.

6.3.4 Logging Results

Logging results is necessary to record and evaluate the performance during benchmarking but it is important that the process of logging results does not, itself, distort the benchmarking process and also that it accurately records the events in the order in which they happen.

Making sure that the logging process does not interfere with the benchmark is relatively trivial as it requires the benchmarking to be timed but for the logging process to happen outside this timing loop, as shown in the pseudo-code snippet below. What is less trivial is the possibility of synchronising the benchmarking output with additional output (i.e. from the Java Virtual Machine to monitor garbage collection or Just-in-Time compilation) and making sure that false positives and negatives do not impact the benchmark when logging additional events.

The issue of synchronisation of program output during logging is complicated by the options available to output results in java. Some of these methods include:

- Textual output can be sent to the command line, when the Java Virtual machine (JVM) is initialised, using the standard output stream. This output can be redirected on the command line or programmatically using the `System.setOut()` method or if the command line buffer is large enough then the output can be manually copied to a file after the benchmarking process has been completed but this is potentially prone to errors from the user and is time consuming;
- Error messages are also displayed on the command line using the standard error streams, however it is not synchronised with the standard output stream and so if the granularity and synchronisation of result reporting is important then only one stream should be used. If the output is redirected from the command line then the standard error is also redirected but can be redirected separately using the `System.setErr()` method;
- Output from the JVM is typically made to the system console and includes verbose output regarding garbage collection and just-in-time compilation, if enabled with the appropriate command line flags. The system console is, by default, created when the JVM is started from the command line if, and only if, the standard output and error streams are not redirected. The behaviour of the system console when output streams are redirected is dependent on the platform but the typical behaviour is that the console is not initialised and the optional JVM output is not available;

- The JVM provides an option (`-Xloggc`) to log garbage collection information directly to a file, complete with timestamps. However, these timestamps are generated using the “wall” clock and have millisecond (or, dependent on platform, tens of millisecond) accuracy; when sub-millisecond accuracy is required this makes it difficult to synchronise two log files, especially if `System.nanoTime()` (see section 6.3.1) is used as this method is only useful in measuring elapsed time (as it uses an arbitrary reference point for measurement, set during JVM initialisation or when first invoked, rather than a fixed point in time).
- Results can be output directly to a file, however, this does not output JVM information sent to the console and adds the complexity of synchronising this additional JVM information.
- An integrated development environment (IDE) can be used to manage the command line output. One example of this is the Eclipse IDE [24], which provides an option to redirect the console output (including the standard output and error streams) to a file. This manages the requirement of keeping the console and standard output stream synchronised but adds the overhead of having another application running in the background.

Given these methods and the particular importance of monitoring the impact of garbage collection, it is important that the results can be properly synchronised and while using an IDE may add additional load on the processor it significantly simplifies the analysis of the results.

6.4 Analysis Techniques to Reduce the Impact of Unmitigated Factors

Mitigating for the impact of external factors (hardware, other software or JVM issues) will reduce the likelihood of outliers but there will be some factors that either cannot be (fully or partially) mitigated for or have not been identified. These factors will introduce outliers into the dataset and can skew the results.

For large sample sizes, outliers are to be expected as part of the distribution and may provide useful cues as to the shape of the distribution; in this case outliers should be retained within the dataset and the analysis should be robust enough to cope with their presence. When measurement errors occur, such as interruptions due to execution of unrelated software, the resulting data is unrepresentative of the underlying distribution and can cause the results to be skewed.

6.4.1 Robust Curve Fitting Techniques

The overall aim for the benchmarking process is to test the hypothesis that the algorithm runs in a given (linear) time as the graph size increases; this means that during post-benchmark analysis it is the upper bound of the time complexity that is relevant in testing this hypothesis. With this in mind, if the upper bound of the time complexity introduced by any unmitigated factors has a greater exponent than the algorithm’s execution time complexity then the unmitigated factor will dominate the analysis rather than the algorithm. There is little that can be done to separate these factors without additional timing information so the benchmarking process is reliant on the fact that any unmitigated factors must have a time complexity that is the same or better than the algorithm’s.

The other issue with unmitigated factors is the issue of intermittent interruptions (such as from garbage collection) that will introduce extreme values into the results; this can either indicate

an erroneous reading (such as from garbage collection) or as part of a distribution with a heavy tail, in which case assumption that the distribution is part of a normal curve and removal of the extreme values as outliers can distort the results. It is necessary to test the shape of the distribution before treating results as outliers.

For simple curves (polynomial, logarithmic, exponential, etc), ordinary linear least squares (OLS) regression [37, Pg. 137-139] provides a method of mathematically calculating the coefficients of a curve so as to find the best fit. OLS works by considering the coefficient of each term in the design model used to specify the curve; minimising the sum of the square of the residuals generates a series of simultaneous linear equations which can be solved to calculate the 'best fit' coefficients.

OLS will find a single, stable solution to the problem; however, it is not resistant to the presence of large outliers, particularly at extreme x values where the presence of an outlier can greatly skew the coefficients of the fitted curve. Figure 6.4.1 shows 20 data points fitted to the $y = 2x^3 - 38x + 19$ curve for x values between -10 and 9, the 21st data point is at (10, 0). The best-fit curves for $y = ax^3 + b$ (green), $y = ax^3 + bx^2 + c$ (red, overlaid by the cyan curve) and $y = ax^3 + bx^2 + cx + d$ (cyan) are shown with the equations in the legend with the calculated coefficients; this demonstrates the fact that a single outlier at the extreme of the dataset has an impact such that the red and cyan lines almost overlap and the x^2 term is given greater significance than the x term in the last equation when it is not in fact the case.

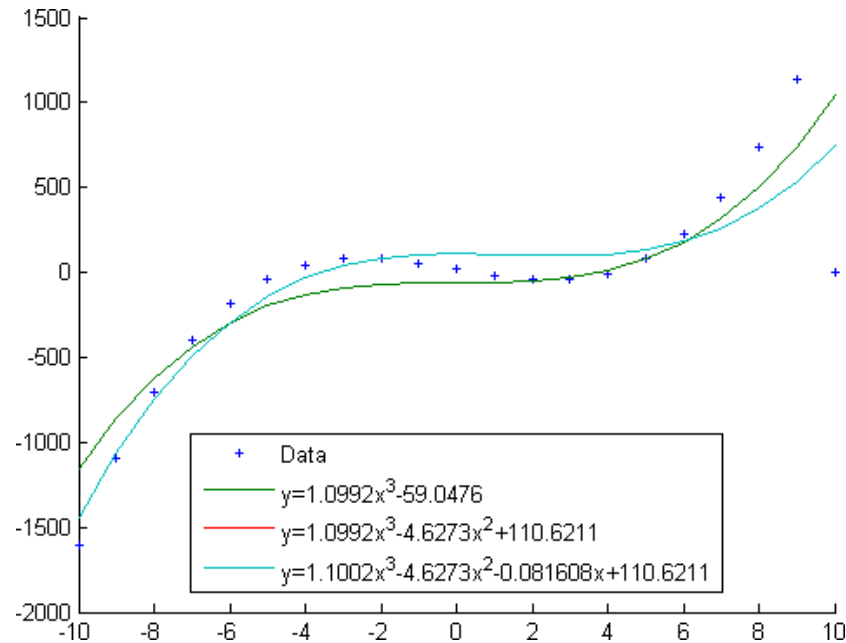


Figure 6.4.1: Least Squares Curve Fitting

There are various alternatives to OLS that provide a better fit to the curve in the presence of outliers. One of these is to use a weighting on the data to give less importance to outlying data points; there are various weighting functions that can be used, including squared errors, absolute errors, Winsorized errors or Tukey's biweight (or bisquare) function. The last two weighting functions are used to trim or taper the errors such that the greatest outliers have a reduced effect on the regression analysis.

In addition to weighting the data, the process can be improved iteratively; an initial estimate of

the curve fit is given using OLS, then a weighting function is applied with a scale factor to reduce the impact of outliers and subsequent iterations are used to adjust the scale factor to improve the quality of the fit. This method is known as Iteratively Re-Weighted Least Squares (IRWLS) [60, Pg. 818-824] and typically gives an improved fit to the data in the presence of outliers; this is visually demonstrated in Figure 6.4.2, displaying best-fits to the same three curves as before (this time with red and green almost overlapping), which shows that, in all cases, both the outlying data-point and the x^2 term in the best-fit curves are given a lower weighting when compared to Figure 6.4.1.

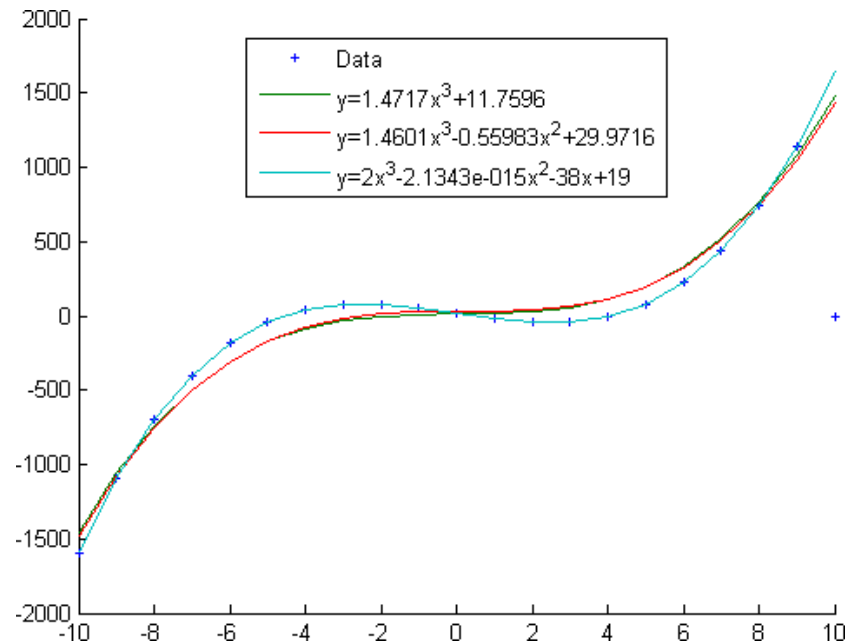


Figure 6.4.2: Iteratively Re-Weighted Least Squares Curve Fitting

When analysing the goodness-of-fit for a design model it is, again, important to use a robust statistic. OLS minimises the square of the residual errors, as such the optimal solution using this method will be identified by the lowest Root Mean Square Error (RMSE); however, as with OLS, the RMSE is greatly affected by the presence of outliers as the sum of the square of many small values can be dwarfed by the square of a single large value. In this case, the Mean Absolute Error (MAE) provides a better [60, Pg. 722-723] estimator for goodness-of-fit, as it is not as skewed by large outliers. This can be seen in Table 6.4.1, where OLS finds the optimum fit to minimise RMSE but IRWLS produces a near exact match to the curve however the RMSE metric indicates that it is the worst fit to the curve; if, instead, the MAE is compared then all the fits produced by IRWLS are an improvement on the OLS solutions and the correct solution is clearly highlighted by the metric.

	Fitted Equation (To 3 S.F.)	RMSE (To 2 D.P.)	MAE (To 2 D.P.)
OLS (Fig. 6.4.1)	$y = 1.10x^3 - 59.0$	284.95	173.97
	$y = 1.10x^3 - 4.63x^2 + 111$	241.51	156.67
	$y = 1.10x^3 - 4.63x^2 - 0.0816x + 111$	241.51	156.59
IRWLS (Fig. 6.4.2)	$y = 1.47x^3 + 11.8$	335.20	146.89
	$y = 1.46x^3 - 0.560x^2 + 30.0$	324.39	142.84
	$y = 2.00x^3 - 2.13 \times 10^{-15}x^2 - 38.0x + 19.0$	357.66	78.04

Table 6.4.1: Comparison of Ordinary and Iteratively Re-Weighted Least Squares

The MAE provides a metric to compare different best fit-models and the lowest MAE indicates the most appropriate fit to the empirical data. Therefore, the use of robust curve fitting techniques (Iteratively Re-weighted Least Squares) and statistics (Mean Absolute Error) will give a better solution in the presence of outliers and in identifying the comparative goodness-of-fit of different design models.

6.5 Benchmarking Methodology Conclusions

Table 6.5.1 summarises the various benchmarking factors detailed earlier in this chapter and the ways that they can be mitigated. The occurrence associated with each factor is an indication of how prevalent that factor is. The severity associated with each factor is an indication of much the factor affects the benchmark compared to the ability to mitigate against its effect; a low severity indicates a factor that either of little impact or that is relatively simple to mitigate against whereas a high severity indicates a factor that has a large impact on the benchmark and is difficult to mitigate against.

Table 6.5.1: Summary of Benchmarking Factors

Factor	Description	Occurrence	Severity	Mitigation
Ordinary Environmental Factors	External effects such as variation in temperature or humidity. Requires specialist equipment to monitor and control.	Rare	Nil – Low	Discounted due to negligible impact and cost of mitigation.
Hardware & Software Setup	Hardware or software setup changes between BMs producing incomparable results.	Rare	Nil – Low	Keep constant hardware and software setup for all BMs.
Clock Resolution	Clock resolution provides minimum limit on valid benchmark times.	Constant	Low	Use <code>System.nanoTime()</code> . Results should only be considered accurate to the resolution of $10\mu\text{s}$.

Factor	Description	Occurrence	Severity	Mitigation
Code Warmup	Interruptions due to class loading and JIT compilation.	During Startup	Low	Run the code for at least 10,000 iterations before logging results. Generate repeated results. Use robust analysis to minimise effect of outliers.
User Interaction	Disruption to benchmark's (BM) execution through user interaction.	Rare	Low	Automate the BM process, removing need for user interaction.
I/O	Logging results adds an I/O overhead to the BM process	Frequent	Medium	Log results between, and not during, BMs. Generate repeated results. Use robust analysis to minimise effect of outliers.
Software Interruptions	Operating System or other processes interrupts BM	Frequent	Medium	Stop all unnecessary applications & services. Generate repeated results. Use robust analysis to minimise effect of outliers.
Garbage Collection	Garbage Collection interrupts BM	Frequent	High	Log GC interruptions using command line option. Synchronise Console and Standard Output Stream using IDE. Discard results tainted by GC. Avoid false negative GC results. Use non-concurrent Garbage Collector. Test graphs with less than 43,200 vertices (100% GC Saturation).
Terminal Environmental Factors	External effects such as power or hardware failure.	Rare	Terminal	Discard terminated BM and repeat.

Chapter 7

Empirical Testing

The aim of this chapter is to present the results of empirical testing of an implementation to demonstrate examples of the linear execution time theorised in chapter 5 and includes:

- Discussion of a Java implementation (source code contained in appendix D) of the planarity testing algorithm;
- Plan for empirical testing of implementation;
- Overview of different graph types, their properties and algorithms for generation;
- Empirical testing results for maximal planar graphs of constant size and varying root vertex;
- Empirical testing results for maximal planar graphs of constant root vertex and varying size;
- Empirical testing results for biconnected sub-graphs of maximal planar graphs of constant root vertex and number of vertices and varying number of edges; and
- Empirical testing conclusions.

7.1 Implementing the Planarity Testing Algorithm

Appendix D contains Java source code for an implementation of the algorithm used later in this chapter to test efficiency of the algorithm.

The source code is separated into several groups of packages:

1. The core classes related to the planarity testing and embedding algorithm. This includes:
 - The basic data structures relating to lists and permutations (appendix section D.1, page 240) used throughout the planarity test;
 - The core graph classes (Block, Edge, Graph, Path, Segment and Vertex) containing the implementation of the planarity testing and embedding algorithm (appendix section D.2, page 268); and
 - Classes relating to the generation of graphs with specific properties (appendix section D.3, page 327).
2. The test harness used to generate specific benchmarks during empirical testing and ancillary utilities to process the log files (appendix section D.4, page 344).
3. A Graphical User Interface for visualisation of graphs and their segment and block properties (appendix section D.5, page 363). This is not required for the scope of the empirical testing but is presented, for completeness, as a method of visually inspecting the properties generated during an embedding and of manipulating embeddings by changing segment permutations and block flips.
4. A utility class (appendix section D.6, page 419) to generate a graphical (L^AT_EX PGF/TikZ) representation the segment hierarchy as used in section 7.3.8 (page 138); again, this is not required for the scope of the empirical testing but is presented for completeness.

7.1.1 Aims and Limitations of the Planarity Testing Algorithm

The aim of this testing is to demonstrate the linear execution time of the software in the case of biconnected graphs and, correspondingly, the focus of the implementation is on biconnected (and not connected graphs). As such, the capability provided by the source code includes:

- Tests for planarity by generating an embedding of a connected graph;
- Identifies *static* segments;
- Identifies *flippable* blocks which define a *Whitney flip* within an embedding;
- Identifies groups of *permutable* segments;
- A method for *flipping* blocks and *permuting* groups of *permutable* segments;
- A method for generating a cyclic edge order given a fixed *flippable* status for each block and a fixed order for each group of *permutable* segments.

However, since the algorithm's focus is on testing linear execution time, it's capability for handling separable graphs is limited to specific functionality where a linear execution time is theorised to exist. In particular, it can generate all possible embeddings of each biconnected component but does not consider the multitude of arrangements when combining the embeddings of those separable components, as the process of re-arranging those components does not have a well defined linear-time implementation, and instead a single embedding of those combined components is generated (which can be performed in linear-time); i.e. the effect of manipulating the *rotatable*, *reversible* and *outer reversible* nature of segments in each biconnected component is not considered.

7.2 Empirical Testing Plan

Chapter 5 presents an algorithm to test a (bi)connected graph for planarity of by generating an embedding of that graph. As part of the generation process, the algorithm constructs a data structure representing possible permutations of the embedding by identifying *Whitney flips* or *permutations* of segment order. Two additional phases of the algorithm are presented to: reorder the data structures representing the permutations of the embedding; and, given an order of those data structures, to generate an cyclic order for the edges incident to each vertex.

Considering the input variables to the algorithm, the following can be varied:

- The graph; this can be further considered in terms of:
 - The graph’s structure;
 - The number of vertices of the graph;
 - The number of edges of the graph; or
 - The initial order for edges incident to each vertex.
- The choice of root vertex.

To give the testing an upper bound, a limiting case can be identified when the graph has the maximum possible edges per vertex without being non-planar. Multi-graphs can be excluded (mainly in the interests of simplicity and expedience), since every multi-graph can be reduced to an equivalent minor by deleting duplicate edges so if that minor is planar then the multi-graph is also planar and, conversely, if multi-graphs were considered then no upper bound can be placed on the number of edges. Given this constraint, then limiting case occurs when the graph is maximal planar.

In a similar vein, chapter 6 identified that at approximately 15,000 vertices then ~25% of the results would be tainted by garbage collection and should be discarded. Therefore, an upper bound can be placed on the graph size to limit testing to graphs of this size otherwise excessively large data sets would need to be generated to ensure a reasonable sample size.

Given these two facts then a bound is also implicitly placed on the number of edges for a biconnected graph between the minimum (where the edges form a Hamiltonian cycle and there are equal numbers of edges and vertices) and a maximal planar graph where there are $(3\mathcal{V} - 6)$ edges.

The graph’s structure can be considered, initially, in terms of 3-, 2- or 1-connected graphs:

- 3-connected graphs have a unique embedding¹ and a particularly interesting sub-set of this family of graphs are the maximal planar graphs (simple planar graphs such that the addition of another edge to the graph makes it either non-simple or non-planar)². Maximal planar graphs are particularly interesting as (a) the number of vertices is proportional to the number of edges ($|\mathcal{E}| = 3|\mathcal{V}| - 6$) and, since a maximal planar graph is biconnected then all leaves of the Trémaux Tree are cotree edges hence the number of cotree edges (and segments) is $(2|\mathcal{V}| - 5)$ (and proportional to the number of vertices); and (b) it is trivial to generate maximal planar graphs of increasing size.

¹ Corollary to Whitney’s 2-Isomorphism Theorem [81, pg. 246].

² All maximal planar graphs have a planar embedding on the surface of a sphere and, by using flat faces rather than following the surface of the sphere, is also the 1-skeleton of a convex polyhedron (3-dimensional convex polytope); by Steinitz’s Theorem [27, Pg. 268] “A graph is 3-polytopal if and only if it is planar and 3-connected.”, it follows that maximal planar graphs are 3-connected.

- 2-connected (biconnected) graphs can be partitioned into two sub-categories³ by considering their minor graphs after coalescing each degree two vertex with an adjacent neighbour; the resulting minor is either 3- or 2-connected.
 - In the case that the resulting minor is 3-connected then the original graph has a unique embedding; and
 - In the case that the minor is 2-connected then at least one 2-separation exists and the embedding can be permuted via *Whitney flip(s)* to give alternate embeddings.
- 1-connected (separable) graphs are not being considered for this purpose as a data structure representing all their possible embeddings does not appear to be able to be generated in linear time.

Various abstract characteristics can also be considered:

- *Flips* and *permutations* – since one of the important aspects of the algorithm is the ability to generate data structures representing all possible embeddings of a biconnected graph it is necessary to test graphs that permit multiple embeddings. By necessity, for a graph to exhibit *flips* or *permutations* then it cannot be 3-connected so some cases of 2-connected graph must be considered.
- The variation in the degree of the graph's vertices – this presents two obvious extremes: when the variance in the vertex degree is as low as possible (and the vertices all have similar degrees); and when the variance in the vertex degree is higher and one or more vertices have a high degree and the rest have low degrees. Relating this to segments, then graphs with a vertex with high degree will have a large number of segments incident to that vertex
- Another characteristic that can be considered is randomness – graph with specific properties will test those properties and, for the most part, only those properties; however, random graphs have the potential to highlight unknown issues. The converse is also true as, when specific properties are required (such as *permutations* in an embedding) it is difficult guarantee such a property through randomness without resorting to a large search space.

From this four types for graph can be distilled:

1. A graph with low variation in vertex degrees;
2. A graph with high variation in vertex degrees;
3. A graph with a range of vertex degrees; and
4. A graph with random vertex degrees.

Section 7.3 gives four maximal planar graph types: Triangular Prism (page 131); Planar Wheel (page 132); Ordered Face Trisection (page 133); and Random Face Trisection (page 134), that correspond to the types above as well as algorithms for their generation. These graph types are used throughout the chapter to test the algorithm's performance.

As previously noted though, there is a need for biconnected graphs to test the performance of the algorithm when *flippable* blocks and *permutable* groups of segments are present. This can be achieved by removing edges from the graphs of the above types and, while this is likely to result in some *flippable* blocks, it is difficult to achieve an algorithm to remove edges in such a way as to leave pairs of segments that can be *permuted* (since those segments must have identical head and tail vertices and not conflict with either's descendants). Therefore, there is an additional need to generate biconnected graphs that contain a *permutable* group of segments.

³ Assuming that the graph being considered is not a Hamiltonian cycle.

The final two points for consideration relating to the algorithm's input are: the order in which a graph's edges are input to the algorithm; and the choice of root vertex. Both these factors affect the generation of the Trémaux Tree but, of the two, the edge order has the greater number of permutations and a much more subtle effect; since varying the order of $\prec^*$ related edges does not affect the Trémaux Tree and it is only when the order of $\parallel^*$ related edges is varied that a different $<^{**}$ order (and, correspondingly, $<^s$ segment precedence order) occurs. Varying both these factors is counter-productive since they both produce similar effects and so attributing any changes would be difficult if both varied. Given this, then a determination was made to focus on changes in root vertex as this would always result in a different Trémaux Tree.

Chapter 6 gives details of some of the factors that affect benchmarking software in Java and how to mitigate against them. Putting together the requirements for different graph types and the methodology for implementing a benchmark lead to the construction of a test harness which is included in Appendix D.4.1 (page 344) and the `timePlanarityTest` method it references within the `Graph` class (Appendix D.2.3, page 279). This test harness identifies over 100 combinations of graph type, graph size and root vertex that test different aspects of the algorithm.

The remainder of this chapter covers two main areas:

1. The properties of the graphs used in these benchmarks; and
2. The analysis of the results obtained by executing these benchmarks.

7.3 Test Graphs

The following section presents six methods for generating types of graphs designated as:

- Triangular Prism Graphs;
- Planar Wheel Graphs;
- Ordered Face Trisection Graphs;
- Random Face Trisection Graphs;
- Permutable, Flippable Graphs; and
- Sub-Graphs Generation via Non-Bridge Edge Deletion.

The first two, Triangular Prism and Planar Wheel graphs, were chosen as they:

- Are well-defined (the graphs can be built in a predictable, repeatable manner);
- Are scalable (when a series of graphs of the same type and increasing size are compared then any smaller graph is a minor of a larger);
- Are maximal planar (thus are triconnected and have a linear relationship between vertex-, edge- and segment-size); and
- Have polar opposite characteristics relating to vertex degrees and maximum shortest path (as described in 7.3.7 on page 137).

The third graph, designated as an Ordered Face Trisection Graph, is presented as a counterpoint to the previous two as it is also maximal planar and generated in a well-defined, scalable manner. However, compared to the previously mentioned graph types, the characteristics of the graph as graph size increases are not as simple to identify; for example, the frequency of vertex degrees does not follow a trivial pattern as graph size increases and the simple repeatable segment structures that are presented for the other graph types are not as readily apparent for this graph structure; therefore while the graph shows repeated structures it is difficult to show repetitions in the structure associated with a linear increase in graph size. This is a useful counterpoint to the previous two graphs and tests the algorithm with a well-defined graph (so that the results are repeatable) but without the linear changes to the structure of the graph that are present in the other graph types.

Random Face Trisection Graphs are also maximal planar but are generated in a random manner, in contrast to the well-defined manner that Ordered Face Trisection Graphs are generated in; this allows graph structures to be tested that might not otherwise be present in the previous, well-defined, graphs.

The Permutable, Flippable Graph type is presented as a method of generating biconnected graphs which contain: a large group of segments which are *permutable*; and a large number of segments contained in *flippable* blocks (as the name suggests). These graphs are not maximal planar but they are well-defined, scalable and where the sub-graph generation algorithm discussed below may generate these properties it cannot guarantee it so this graph type aims to fill that void.

The final graph generation method considers non-maximal planar graphs – taking a graph of one of the previous four maximal planar types then generating a (scalable) series of connected sub-graphs of constant vertex- and decreasing edge-size can be generated via deletion of successive non-bridge edges from that graph; this allows for the generation of biconnected and separable graphs. By specifying an order to the selection of edges for deletion then the sub-graphs can be generated in a well-defined manner from any given super-graph.

7.3.1 Triangular Prism Graph

The first graph presented is a Prism Graph [25], given by the Cartesian Graph Product $(C_3 \times P_{\lfloor \frac{N}{3} \rfloor})$, extended with additional edges connecting sequential concentric triangles. When viewed in perspective with a central vanishing point then the graph appears to form a triangular prism.

A triangular prism graph, of size N (where $k = \lfloor \frac{N}{3} \rfloor$, $r = \text{modulo}(N, 3)$ and $i = \min(1, r)$), is formed of k concentric equilateral triangles of vertices and r additional vertices (either zero, one additional vertex at the centre or one vertex at the centre and one outside). The vertices are ordered in ascending numerical order from the centre spiraling outwards such that:

- If $i = 1$, Vertex V_1 is at the centre of the graph.
- A radial line through vertex V_{1+i} also passes through $V_{4+i}, V_{7+i}, \dots, V_{3(k-1)-2+i}, V_{3k-2+i}$;
- A radial line through vertex V_{2+i} also passes through $V_{5+i}, V_{8+i}, \dots, V_{3(k-1)-1+i}, V_{3k-1+i}$;
- A radial line through vertex V_{3+i} also passes through $V_{6+i}, V_{9+i}, \dots, V_{3(k-1)+i}, V_{3k+i}$; and
- If $r = 2$, Vertex V_{3k+2} is outside of the outermost concentric triangle on the radial line passing through V_2 .

The edges are created in the order:

1. For $p = 2 \dots (3 + i)$, for $q = 1 \dots (p - 1)$,
 - (a) Edge (V_q, V_p) is created. [Innermost $\triangle$ (and centre vertex if required)]
2. For $p = 2 \dots k$
 - (a) $(V_{3p-2+i}, V_{3p-1+i}), (V_{3p-2+i}, V_{3p+i}), (V_{3p-1+i}, V_{3p+i})$ [Edges forming the next concentric triangle]
 - (b) $(V_{3p-5+i}, V_{3p-2+i}), (V_{3p-3+i}, V_{3p-2+i})$ [Connecting 1st vertex of $\triangle$ to previous $\triangle$]
 - (c) $(V_{3p-4+i}, V_{3p-1+i}), (V_{3p-5+i}, V_{3p-1+i})$ [Connecting 2nd vertex of $\triangle$ to previous $\triangle$]
 - (d) $(V_{3p-3+i}, V_{3p+i}), (V_{3p-4+i}, V_{3p+i})$ [Connecting 3rd vertex of $\triangle$ to previous $\triangle$]
3. If $r = 2$, $(V_{N-3}, V_N), (V_{N-2}, V_N), (V_{N-1}, V_N)$ [Connecting last vertex to previous $\triangle$]

This can be visualised as:

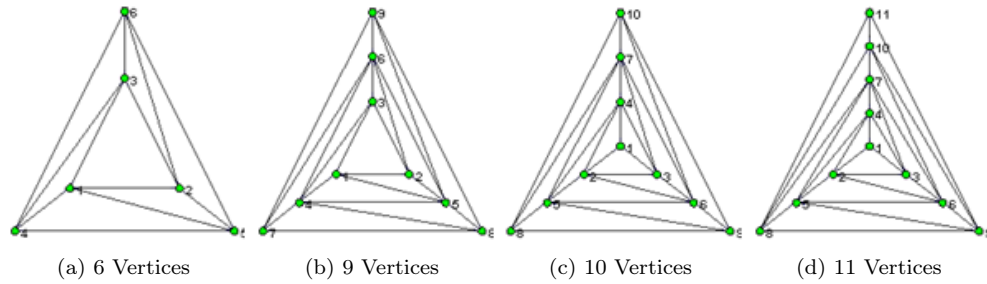


Figure 7.3.1: Examples of Triangular Prism Graphs

Triangular Prism Graphs have the following frequencies of vertex degrees:

modulo(N , 3)	Frequency of Vertices of:			
	Degree 3	Degree 4	Degree 5	Degree 6
0	0	6	0	$N - 6$
1	1	3	3	$N - 7$
2	2	0	6	$N - 8$

Table 7.3.1: Vertex Degree Frequencies in a Triangular Prism Graph of size N .

7.3.2 Planar Wheel Graph

This graph design is formed by three overlapping Wheel [25] Sub-Graphs (designated as W^1 , W^2 and W^3). The graph can be described by: one central vertex (present in all wheel sub-graphs); three paths radiating from the central vertex (each path shared by two wheel sub-graphs); three hub vertices (unique to each wheel sub-graph) with connecting edges (spokes of the wheel) to the central vertex and each vertex of the two adjacent radial paths; and three edges, each unique to a wheel sub-graph and completing a cycle with two radial paths and the central vertex.

For a planar wheel graph, of size N , the wheel sub-graph (W^i) comprises of (where $i = 1 \dots 3$ and $j = \text{modulo}(i + 1, 3) + 1$):

- The cycle of vertices $\langle V_1, V_{(\lceil \frac{(i-1)(N-1)}{3} \rceil + 3)}, \dots, V_{(\lceil \frac{i(N-1)}{3} \rceil + 1)}, V_{(\lceil \frac{j(N-1)}{3} \rceil + 1)}, \dots, V_{(\lceil \frac{(j-1)(N-1)}{3} \rceil + 3)} \rangle$; and
- The hub vertex $V_{(\lceil \frac{(i-1)(N-1)}{3} \rceil + 2)}$

The edges are created connecting, in the order:

1. For $i = 1 \dots 3$:
 - (a) Hub vertex $V_{(\lceil \frac{(i-1)(N-1)}{3} \rceil + 2)}$ to the central vertex V_1 ;
 - (b) Innermost spoke vertex $V_{(\lceil \frac{(i-1)(N-1)}{3} \rceil + 3)}$ to the central vertex and the hub vertex;
 - (c) For $j = (\lceil \frac{(i-1)(N-1)}{3} \rceil + 4) \dots (\lceil \frac{i(N-1)}{3} \rceil + 1)$:
 - i. The spoke vertex V_j to the previous spoke vertex V_{j-1} ; and
 - ii. The spoke vertex V_j to the hub vertex.
2. For $i = 1 \dots 3$, create an edge such that it joins:
 - (a) The $\text{modulo}(i, 3) + 1^{\text{th}}$ hub to each vertex in the i^{th} spoke (from centre out); and
 - (b) The vertices terminating the i^{th} and $\text{modulo}(i, 3) + 1^{\text{th}}$ spoke.

This can be visualised as:

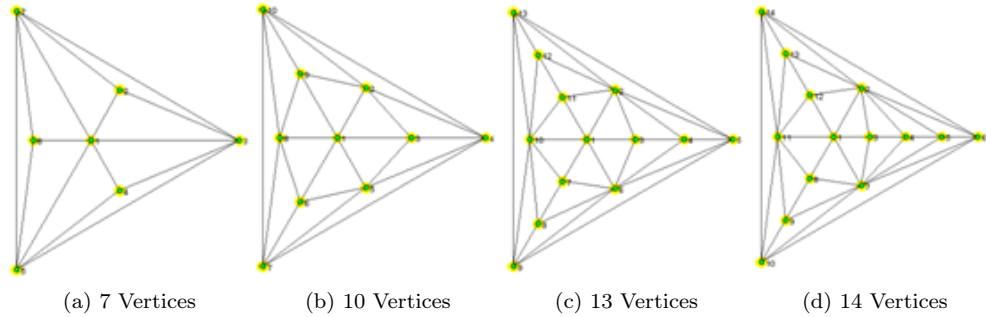


Figure 7.3.2: Examples of Planar Wheel Graphs

Planar Wheel Graphs have the following frequencies of vertex degrees:

modulo(N , 3)	Frequency of Vertices of Degree:							
	4	5	6	$\frac{2N-3}{3}$	$\frac{2N-4}{3}$	$\frac{2N-5}{3}$	$\frac{2N-6}{3}$	$\frac{2N-7}{3}$
0	$N - 7$	3	1	1	0	0	2	0
1	$N - 7$	3	1	0	0	3	0	0
2	$N - 7$	3	1	0	2	0	0	1

Table 7.3.2: Vertex Degree Frequencies in a Planar Wheel Graph of size N .

7.3.3 Ordered Face Trisection Graph

This graph starts with a triangular cycle formed by three vertices and three edges creating an internal and an external face; this internal face is placed in a queue of faces. When a vertex is added to the graph: the face, at the head of the queue, is removed; the new vertex is placed at the centre of that face; that face is trisected by three edges connecting the vertices at the boundary of the face with the new vertex; and the three new (triangular) faces formed by this trisection are added to the tail of the queue of faces.

The edges are created in the following order:

1. The initial triangular face is created containing (in the order created):
 - (a) Vertices v_1, v_2 and v_3 ; and
 - (b) Edges $\{v_1, v_2\}, \{v_1, v_3\}, \{v_2, v_3\}$.
2. Adding each successive vertex v_K to the graph then the face at the head of the queue (containing vertices, in a clockwise order, v_A, v_B and v_C) will be trisected by the addition of edges $\{v_A, v_K\}, \{v_B, v_K\}$ and $\{v_C, v_K\}$ forming the faces containing (in the order appended to the queue and with vertices listed in a clockwise order):
 - (a) Vertices v_A, v_B & v_K ;
 - (b) Vertices v_B, v_C & v_K ; and
 - (c) Vertices v_C, v_A & v_K .

This can be visualised as:

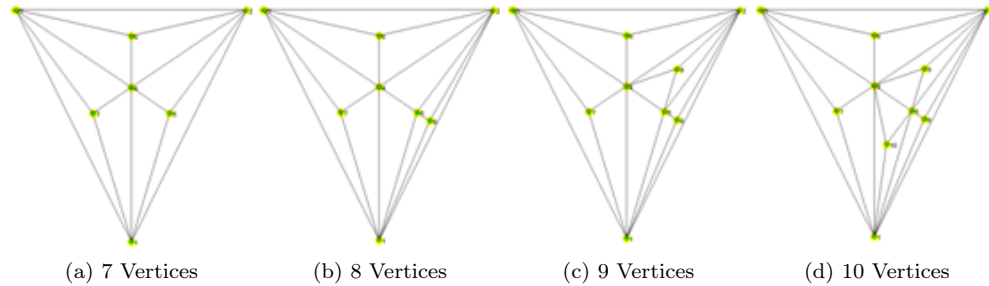


Figure 7.3.3: Examples of Ordered Face Trisection Graphs

Unlike the previous two graphs, this graph does not have a trivial pattern to the frequency of vertex degrees. For graphs with N vertices, there will be: exactly $\lfloor \frac{2N-5}{3} \rfloor$ vertices of degree 3 (for $N \geq 6$); approximately $\frac{2}{9}N$ vertices of degree 6; approximately $\frac{2}{27}N$ vertices of degree 12; and, generally, approximately $\frac{2}{3^{k+1}}N$ vertices of degree (3×2^k) , where $k \in \mathbb{N}_0$. There may be individual vertices with degrees outside the previous pattern. For example, when $N > 12$, if $\text{modulo}(N, 3)$ is 0 then the graph contains one vertex of degree 4 but when the modulus is 1 there exists one degree 5 vertex and when the modulus is 2 the graph will not contain any degree 4 or 5 vertices; similar, but not as simple, patterns exist for vertices within the general range of degrees $(3 \times 2^k + 1)$ through $(3 \times 2^{k+1} - 1)$ for $k > 2$.

7.3.4 Random Face Trisection Graphs

This graph starts with a triangular cycle formed by three vertices and three edges creating an internal and an external face; this internal face is placed in a (empty) array of faces. When a vertex is added to the graph: a face is removed from a random position in the array; the new vertex is placed at the centre of that face; that face is trisected by three edges connecting the vertices at the boundary of the face with the new vertex; and the three new (triangular) faces formed by this trisection are added to the array – one replacing the removed face and the other two faces added at the end of the array.

Since faces are randomly selected for trisection then any pattern in the degrees of vertices as the graph size increases is coincidental to the order in which faces are created. This is a counterpoint to the previous types of graphs, which are created in a rigid manner and, correspondingly, have a distinct pattern to the structure of the graph and the degrees of the vertices.

7.3.5 Permutable, Flippable Graphs

A permutable, flippable graph, unlike the previous graphs is not maximal planar; instead it is designed to contain a group of *permutable* segments (*permutable* with the parent segment and containing its *static* child) and a number of *flippable* blocks.

It can be constructed such that:

- Vertices v_1 , v_2 and v_3 are connected by edges $\{v_1, v_2\}$ and $\{v_1, v_3\}$; and
- Each subsequent vertex v_n is connected by edges $\{v_2, v_n\}$ and $\{v_3, v_n\}$ and if (modulo $(n, 2) = 0$) edge $\{v_{n-1}, v_n\}$.

The graph appears similar to a gemstone where pairs of successive vertices (after the first three) form facets of the stone. Each facet (excluding the root segment) is composed of a parent segment, which is part of a *permutable* group of segments, and its descendants, which form a *flippable* block.

For a *permutable, flippable* graph of n facets there are $n!/2$ *permutations* of segments (since one segment is static) and 2^n *flippable* blocks giving a total of $n!2^{(n-1)}$ possible embeddings of the graph.

This can be visualised as:

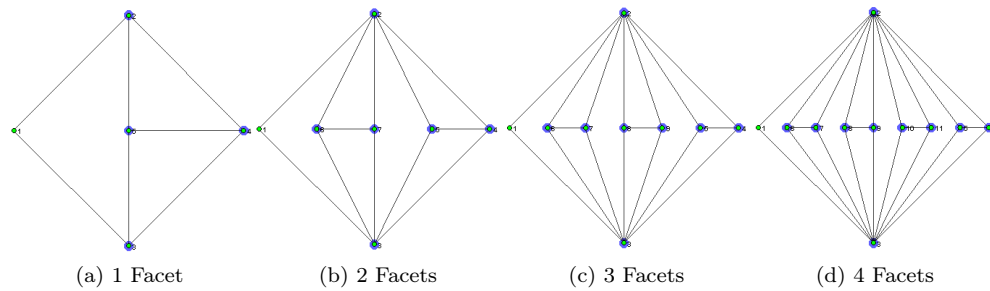


Figure 7.3.4: Examples of Permutable, Flippable Graphs

7.3.6 Sub-Graph Generation via Non-Bridge Edge Deletion

The previous graph types (ignoring Permutable, Flippable graphs) have focused on maximal planar graphs and, as such, the number of vertices is proportional to the number of edges in each of those graphs. An algorithm to delete successive edges from a given (maximal planar) graph while maintaining that the graph remains connected gives a method of testing the performance of the planarity testing algorithm against graphs of a constant number of vertices but varying numbers of edges.

Given a planar embedding Γ of graph $\mathcal{G}$ then any non-bridge edge e is contained in at least one cycle (lemma 4.1.19) and, by the Jordan Curve theorem [30, 39], is the boundary between two disjoint faces (f_1 and f_2). Considering $\Gamma \setminus \{e\}$ then the resulting set of faces is $\mathcal{F} \cup \{f_{(1,2)}\} \setminus \{f_1, f_2\}$ where $f_{(1,2)}$ is the region $(f_1 \cup e \cup f_2)$. Repeated application of this operation results in the size of the edge- and face-sets ($|\mathcal{E}|$ and $|\mathcal{F}|$, respectively) reducing linearly and the size of the vertex-set ($|\mathcal{V}|$) remains constant until there are no more non-bridge edges (when $|\mathcal{V}| = |\mathcal{E}| + 1$ and $|\mathcal{F}| = 1$).

This leads to an algorithm to generate planar sub-graphs from a planar graph given a Trémaux Tree of that graph. Removal of any non-bridge (tree or cotree) edge is such that:

- If the removed edge is a cotree edge then a leaf of the Trémaux Tree has been removed and the relative precedence of the other elements of the Trémaux Tree is unaffected; however,
- If the removed edge is a tree edge then the Trémaux Tree is partitioned into two disjoint tree structures (containing those elements succeeding and not succeeding that tree edge); these can be reconnected by re-designating a cotree edge, connecting the disjoint parts, as a tree edge and refactoring the precedence of the elements within the Trémaux Tree.

Different sequences of successively smaller sub-graphs can be achieved by varying the selection and order in which non-bridge edges are removed; since, if necessary, the Trémaux Tree is re-factored at each step then only the existence of bridge edges restricts the choice for which edge is removed and the choice of initial spanning tree does not affect this.

Considering the relationship between the number of segment and edges: each segment contains at least one edge (assuming the graph does not consist of a single vertex); therefore, the maximum bound on the number of segments is equal to the number of edges; however, the number of segments will only equal the number of edges if every edge is either a looping edge or a bridge edge outbound from the root vertex. Therefore, for most graphs, the number of segments is less than the number of edges.

Considering the deletion of a non-bridge edge $(u \rightarrow v)$ where a single leaf of the Trémaux Tree is reachable from vertex u then, it follows that, that leaf must be a cotree edge and can be designated $(x \xrightarrow{c} y)$ (where $y \prec u \prec (u \rightarrow v) \prec (x \xrightarrow{c} y)$), and:

- If $(u \rightarrow v) = (x \xrightarrow{c} y)$ then, post-deletion, u becomes a leaf of the Trémaux Tree and the number of leaves remains constant while the number of edges decreases;
- Otherwise $(u \rightarrow v)$ is a tree edge and, post-deletion, the Trémaux Tree is re-organised such that $(x \xrightarrow{c} y)$ is re-designated as a tree edge and the direction of all edges which had previously succeeded v are reversed so v becomes a leaf of the Trémaux Tree so, at a minimum, the number of segments remains constant. If, in this case, $y \prec u$ then, post-deletion, as well as the re-designated path to v there exists a path of tree edges from y to u and so the number of segments will increase by one as the number of edges decreases.

Therefore, edge deletion does not necessarily reduce the number of segments by one and in some cases can, if the deleted edge is a tree edge, increase the number of segments.

The simplest method of generating graphs by edge deletion is to always remove cotree edges from the embedding until only tree edges remain and the graph is reduced to a spanning tree. In this case:

- Each deletion does not increase the number of segments; and
- The choice of initial spanning tree restricts the edges that can be removed; however, this can be mitigated against by (a) selecting a different root vertex⁴, allowing limited variation in the Trémaux Tree, or (b) changing the order in which edges are processed by the DFS algorithm generating the Trémaux Tree and, hence, the Trémaux Tree structure itself.

Biconnected Sub-Graph Generation via Non-Bridge Edge Deletion

In a similar vein to generating connected sub-graphs, biconnected sub-graphs can be generated through careful choice of edges for deletion.

Considering each cotree edge $(u \xrightarrow{c} v)$ then two cases can be considered:

1. When it is the sole edge within a segment – in this case, if the original graph was biconnected then a cotree edge $(u \xrightarrow{c} v)$ must form a Class 4 segment and so has a parent segment which forms a cycle (of itself or with ancestors) which contains both u and v ; therefore, deletion of $(u \xrightarrow{c} v)$ does not make the sub-graph separable.
2. When it is not the sole edge within a segment – in this case, if the original graph was biconnected and the containing segment has a descendant outbound from u (hence u would not be reduced to a degree of 1) then it *may* be possible to delete $(u \xrightarrow{c} v)$.

Assuming $(u \xrightarrow{c} v)$ is contained in segment s of a biconnected graph then if there exists a descendant of s outbound from u which is either: inbound to a vertex preceding s 's head vertex; or is in a block which contains another descendant of s which is inbound to a vertex preceding s 's head vertex, then there exists a cycle containing all other elements of s and some elements of s 's parent; therefore deleting $(u \xrightarrow{c} v)$ will not induce a 1-separation in the graph. However, if there exists no descendant of s with either property then all descendant segments outbound from u are in flippable blocks (which do not contain segments inbound to v) and so deleting $(u \xrightarrow{c} v)$ would induce a 1-separation in the graph.

This second case requires both an embedding of the graph and a test of descendant segments and, if successful, would result in the segments and segment hierarchy being reorganised; therefore it is not the most efficient approach (more efficient methods may exist but that is not the focus of this work).

Therefore, a simple and efficient method of generating a biconnected graph from a 3-connected graph is to delete only cotree edges where they are the sole edge of their segment. This is used within `ReduceableBiconnectedGraphGenerator.java` (section D.3.7) to generate biconnected graphs of the four previous maximal planar graph types and these sub-graphs are used within the empirical testing later in this chapter (section 7.6, page 159).

⁴ This can be seen in 10 vertex Planar Wheel graphs with root of 1st (figure 7.3.7a) or 10th (figure 7.3.8a) vertex; edge {8, 9} is a tree and a cotree edge for those respective graphs.

7.3.7 Comparison of Test Graph's Properties

The test graphs have, for a graph of size N , the properties shown in Table 7.3.3.

Property	Triangular Prism	Planar Wheel	Ordered Face Trisection	Random Face Trisection
Number of Edges	Maximal Planar ($3N - 6$)	Maximal Planar ($3N - 6$)	Maximal Planar ($3N - 6$)	Maximal Planar ($3N - 6$)
Modal Vertex Degree	6 (when $N > 14$)	4 (when $N > 10$)	3 (when $N > 8$)	Random
Vertex Degree Std. Deviation	$\frac{2\sqrt{6(N-6)}}{N}$ $= O\left(\frac{1}{\sqrt{N}}\right)$	$\frac{\sqrt{4N^3-80N^2+454N-432}}{N\sqrt{3}}$ $= O\left(\sqrt{N}\right)^5$	$O\left(\sqrt{N \cdot \log_2(N)}\right)^6$	Random
Maximum Vertex Degree	$6 = O(1)$	$\frac{2N-3}{3} = O(N)$	$O(N)$	$O(N)$
Maximum Shortest Path	$\left\lfloor \frac{N}{3} \right\rfloor + \text{modulo}(N, 3)$ $= O(N)$	$3 = O(1)$	$O(\log_2(N))$	$O(N)$

Table 7.3.3: Comparison of Test Graph Properties

The triangular prism graph has:

- A constant maximum vertex degree;
- Decreasing standard deviation in the degree of the vertices; and
- A linearly increasing maximum shortest path between pairs of vertices.

Whereas the planar wheel graph has:

- A linearly increasing maximum vertex degree;
- Increasing standard deviation in the degree of vertices; and
- A constant maximum shortest path between pairs of vertices.

Comparing these attributes shows that the two graphs are diametrically opposite in their features and represent two different extremes in structures. The characteristics of the ordered face trisection graph are part way between both extremes; whilst the random face trisection graph is maximal planar, the same as the other types, but has no set properties.

⁵ When $\text{modulo}(N, 3) = 1$; for other graph sizes $\sigma = \frac{\sqrt{4N^3-80N^2+438N-432}}{N\sqrt{3}}$.

⁶ Mean vertex degree ($\bar{x}$) is $(6 - \frac{12}{N})$.

There are approximately $\frac{2}{3^{(k+1)}}N$ vertices of size $(3 \cdot 2^k)$ for $k = 0 \dots \left\lfloor \log_2 \frac{N}{3} \right\rfloor$.

$\therefore \sigma \approx \sqrt{\sum_{k=0}^{\left\lfloor \log_2 \frac{N}{3} \right\rfloor} \left[(3 \cdot 2^k)^2 \left(\frac{2}{3^{(k+1)}}N \right) \right] - \bar{x}^2} = \sqrt{6 \sum_{k=0}^{\left\lfloor \log_2 \frac{N}{3} \right\rfloor} \left[\left(\frac{2}{3} \right)^k 2^k \right] - \bar{x}^2} \leq \sqrt{6 \sum_{k=0}^{\left\lfloor \log_2 \frac{N}{3} \right\rfloor} [2^k] - \bar{x}^2}$
 $\leq \sqrt{6 \left(\log_2 \frac{N}{3} + 1 \right) 2^{\left(\log_2 \frac{N}{3} \right)} - \bar{x}^2} \leq \sqrt{2N (\log_2 N + 1 - \log_2 3) - \left(6 - \frac{12}{N} \right)^2} = O(\sqrt{N \cdot \log_2 N})$

7.3.8 Scalability of Test Graphs

An algorithm for generating graphs is classed as being scalable when two graphs of different size can be generated and the smaller graph is a minor of the larger. For two graphs of the same type, generated using one of the previously listed algorithms (and assuming, for Triangular Prism Graphs, that only full concentric triangles are considered and any central vertex or single vertex in the outer face is ignored), then the graphs are generated such that the smaller graph is always minor of the larger; hence, those graphs can be classed as being scalable.

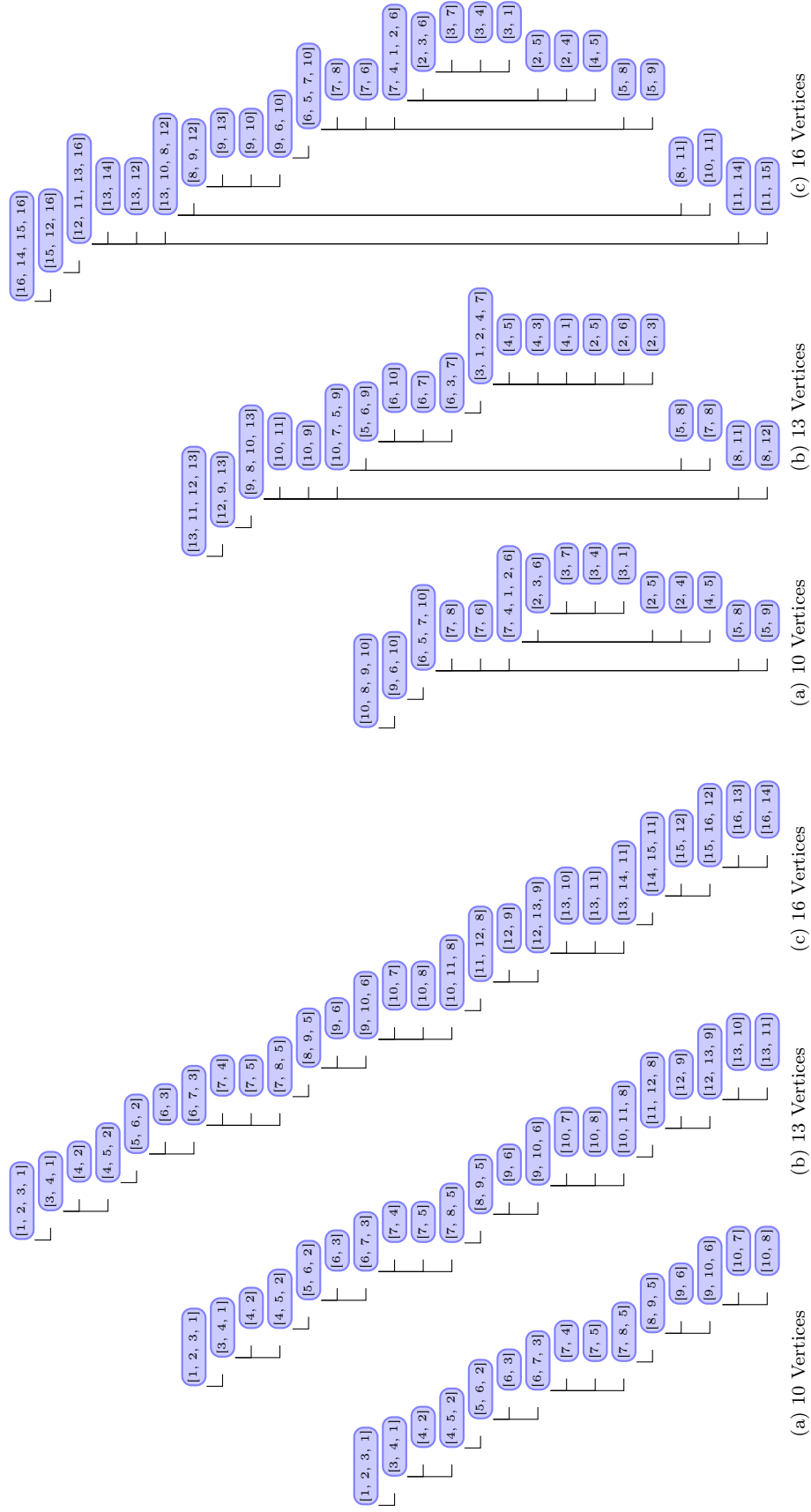
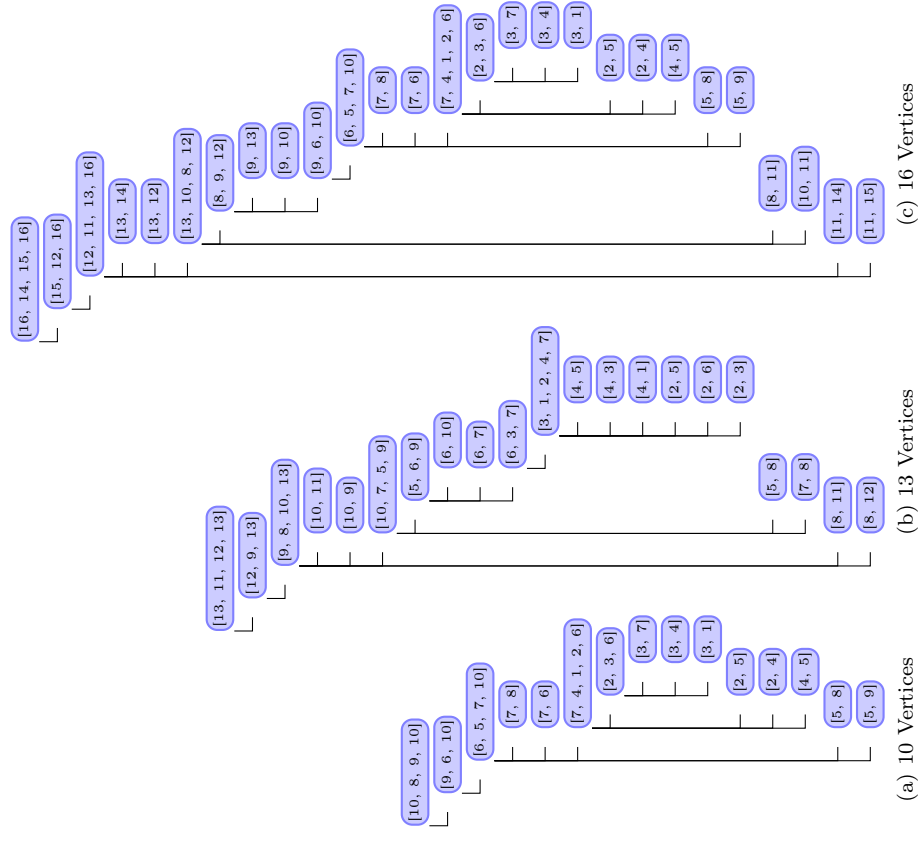
This property can also be investigated using the hierarchy of the segment tree and the composition of those segments: Figures 7.3.7 through 7.3.10 show the segment structures for various graph sizes, types and root vertices; displaying the segments in $<^s$ precedence order and indented to highlight their parent-child relationships. Individual segments are displayed as a bracketed list of vertices starting from the segment's head vertex and terminating at its tail vertex.

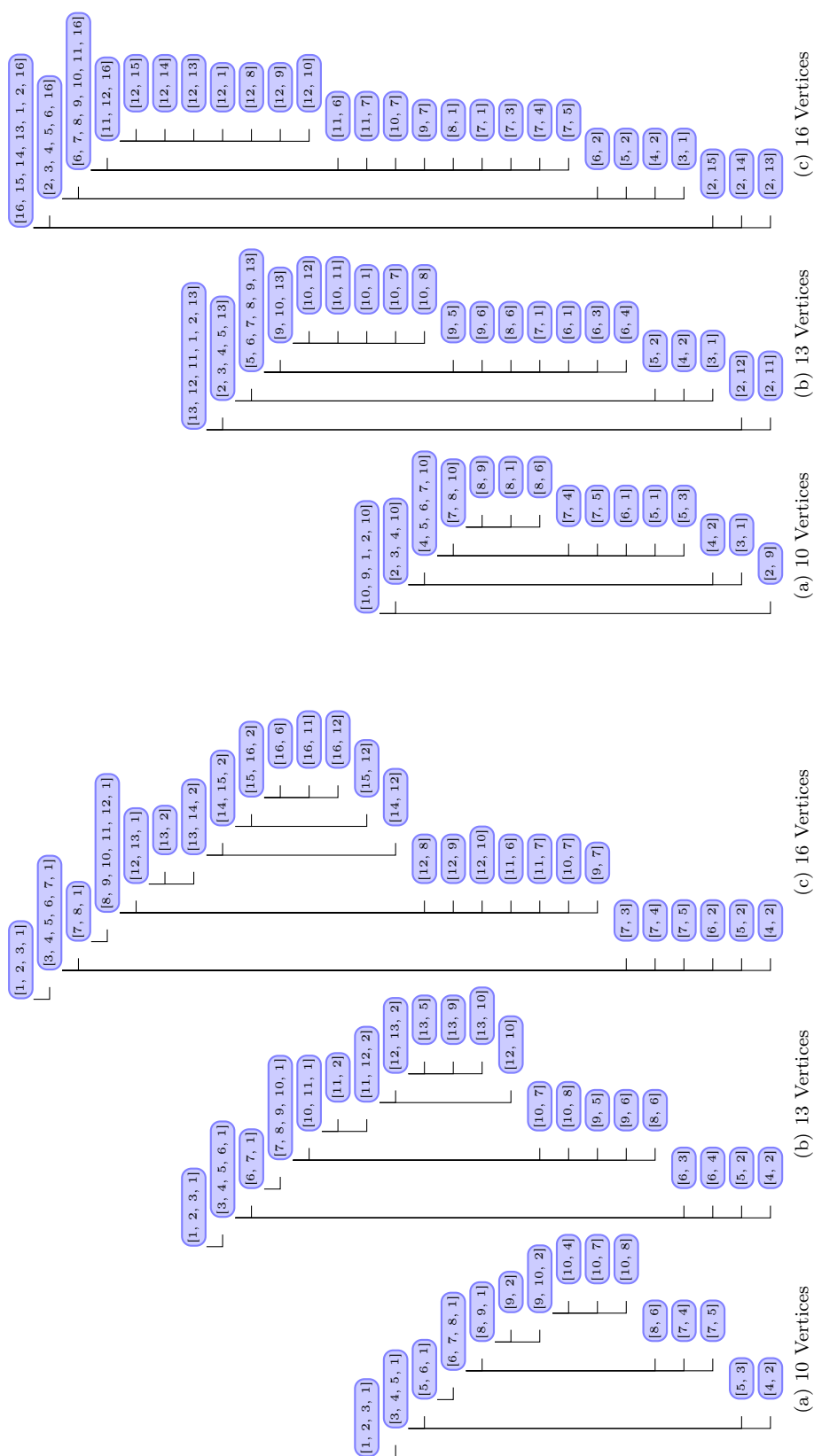
Looking at the graph types and root vertex combinations shows some simple patterns as to how the segment structure evolves:

- The simplest graph segment structure to consider is a Triangular Prism Graph where the root is the 1st vertex (figure 7.3.5); as successive triplets of vertices are added to the graph, the graph's segment structure is a duplication of the segment structure for previous graphs in the sequence with the additional segments, created as the graph size increases, appended to the deepest point of the segment tree. As an aside, for this graph type and root vertex, this means both the mean and maximum segment depths are approximately linear with the graph size.
- In contrast, a Planar Wheel Graph where the root is the last vertex (figure 7.3.8) has a segment tree of constant maximum depth as graph size increases. With each triplet of vertices (and corresponding nine edges) added to the graph: the number of vertices and edges in the first three segments of the hierarchy increases by one; and the remaining six edges all form segments composed of a single cotree edge which are appended in the ratio 1:1:2:2 over, respectively, the 2nd through 5th levels of the segment tree. The average segment depth for this graph type and root is, given this progression and N vertices, $\left(\frac{23}{6} - \frac{9}{4N-10}\right)$.
- For a Planar Wheel Graph with 1st vertex as the root (figure 7.3.7): the 2nd and 4th segments increase in length by one vertex (and edge); the remaining vertex (and two edges) forms a new segment which is appended to the penultimate level of the segment tree, increasing the depth of the tree by one; a new single cotree edge segment is added as a sibling on that penultimate level; and the two pairs of single cotree edge segments are appended at depth two and three in the segment tree.
- A Triangular Prism Graph with the last vertex as root (figure 7.3.6) is similar to when the 1st vertex is the root as segments are appended, in groups, to the deepest segment; however, in this case, the deepest segment is in the middle of the list of segments. From the base structure, recursively, a segment of 3 vertices is added then descending from it a segment of 4 vertices and four segments containing only frond edges (2 vertices); when the graph size increases by three again then the same structure is added again descending from that length 4 segment in the middle of the four length 2 segments.
- The two root vertices shown for the Ordered Face Trisection Graphs (figures 7.3.9 and 7.3.10) both display repeated structures; in particular, when the root is the first vertex then there are groups of segments with depth of 3 which have same structure as the graph

size increases. However, the segment structure is not as easily defined as the other two graph types.

The examples presented are minimal examples, able to be visualised on a single page, and scaling this to larger graphs is impractical to visualise within the constraints of this document; however, it should be noted that similar patterns within the segment structure are present for different root vertices (where the root vertex is chosen as a constant fraction of the total graph size) when the graph sizes increase beyond these minimal examples. The next section gives empirical evidence supporting this claim by comparing the execution time of the algorithm with variation in the root vertex and demonstrating similar patterns in execution time as the graph size increases.

Figure 7.3.5: Segment Hierarchy for Triangular Prism Graphs (Root = 1st Vertex)Figure 7.3.6: Segment Hierarchy for Triangular Prism Graphs (Root = Nth Vertex)



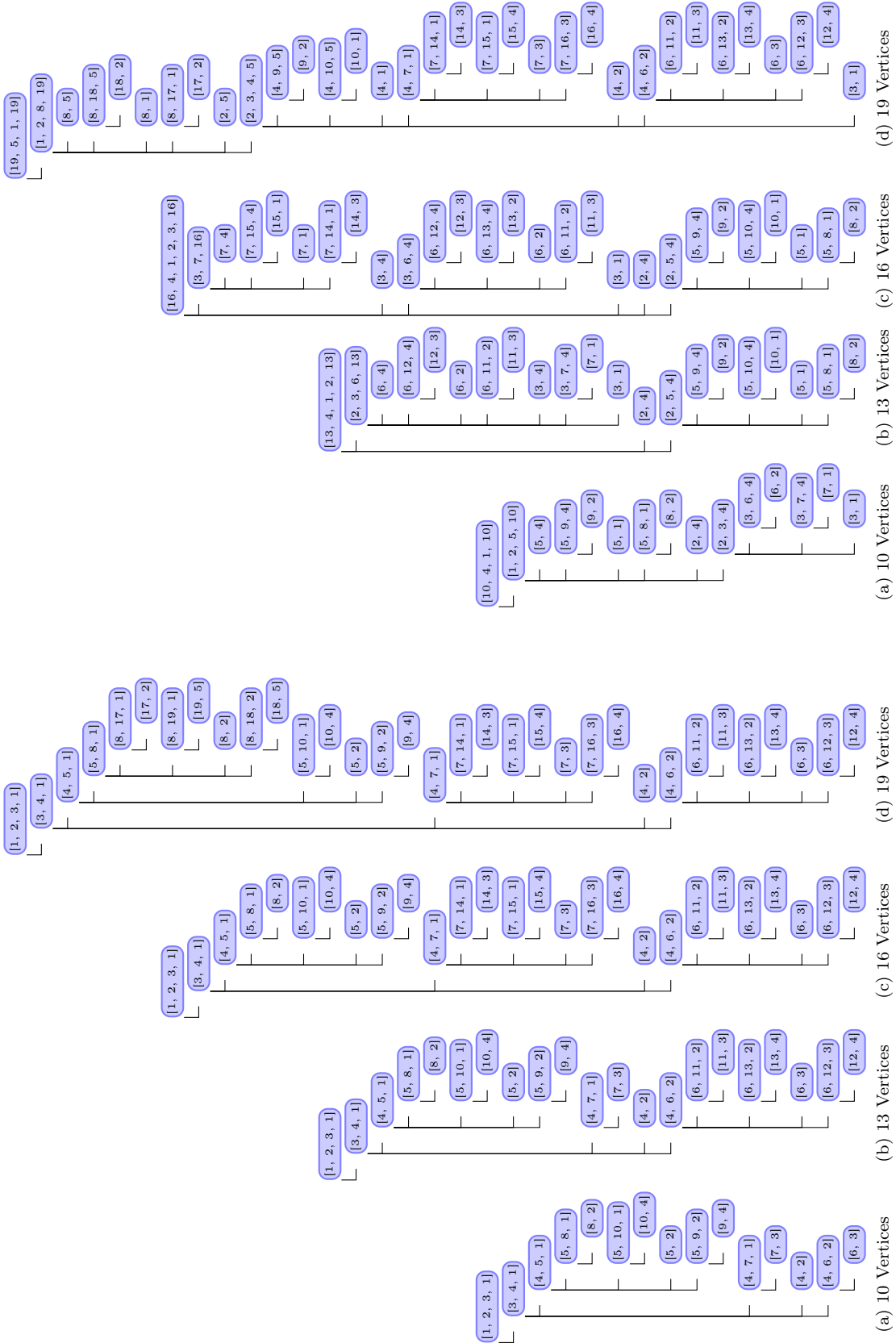


Figure 7.3.9: Segment Hierarchy for Ordered Face Trisection Graphs
(Root = 1st Vertex)

Figure 7.3.10: Segment Hierarchy for Ordered Face Trisection Graphs
(Root = Nth Vertex)

7.4 Empirical Testing using Different Root Vertices

The previous section gives some simple examples of how the segment structure varies as the graph size, structure and choice of root vertex varies. This section expands on the

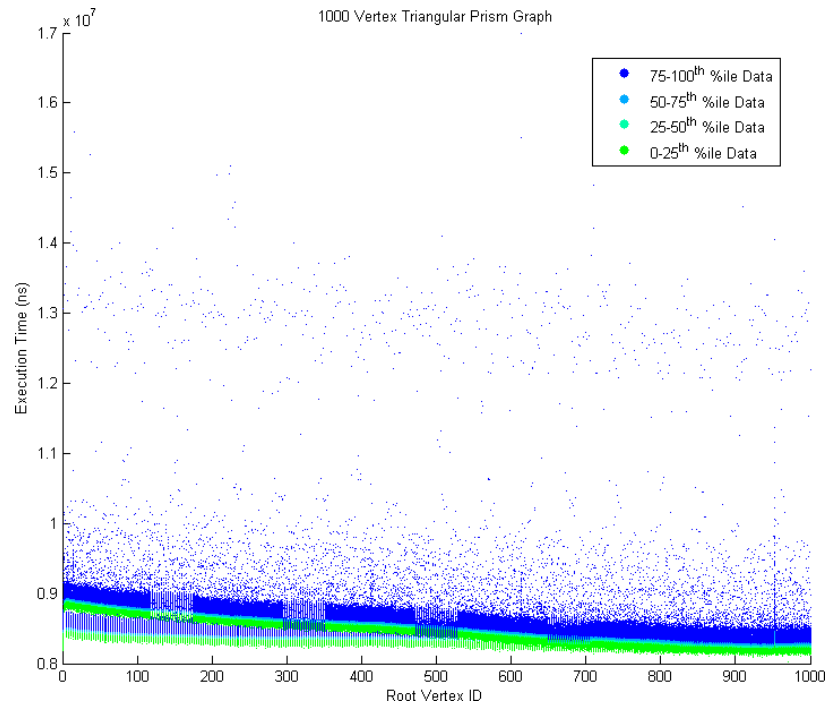


Figure 7.4.1: Execution Times for Varying Root Vertices of 1000 Vertex Triangular Prism Graph

Figure 7.4.1 shows the full data-set from executing the algorithm over a 1000 vertex Triangular Prism Graph and varying the root vertex. The algorithm was executed 1000 times for each root vertex (shown along the x-axis using the sequential numerical id assigned to each vertex as it is created) and the duration of the execution records (y-axis); the execution times are displayed in coloured quartile bands so that the quartile boundaries and interquartile range can be visualised. Several points can be noted regarding the data:

1. The majority of the data is in a band between 8-9.5ms that decreases as the Id assigned to each root vertex increases; however, there are scattered outlying data points up to 100% above this range.
2. There are two distinct curves in the data set that converge as the root vertex Id increases such that: one curve starts (at the lower root vertex id) at median of ~8.9ms; the starts other at ~8.45ms; and both curves decrease to a median of ~8.25ms.

Examining this last point: Vertex ids are assigned in the order that those vertices are created by the algorithm generating the graph and, as such, is an arbitrary numerical value where the relative order is not particularly significant. For a Triangular Prism graph, the graph generation algorithm creates vertices in successive concentric triangles and assigns the vertex id spiraling out from the centre; instead, the vertices can be partitioned into three groups, separated on modulus 3 of their vertex id, representing vertices on a spoke of the graph radiating from its

centre and within those partitions can be re-assigned vertex ids corresponding to increasing distance from the centre. This can be achieved using the mapping $x \mapsto \lceil (N \cdot \text{modulo}(x, 3) + x) / 3 \rceil$, where N is the number of vertices in the graph and x is the default vertex id assigned during generation.

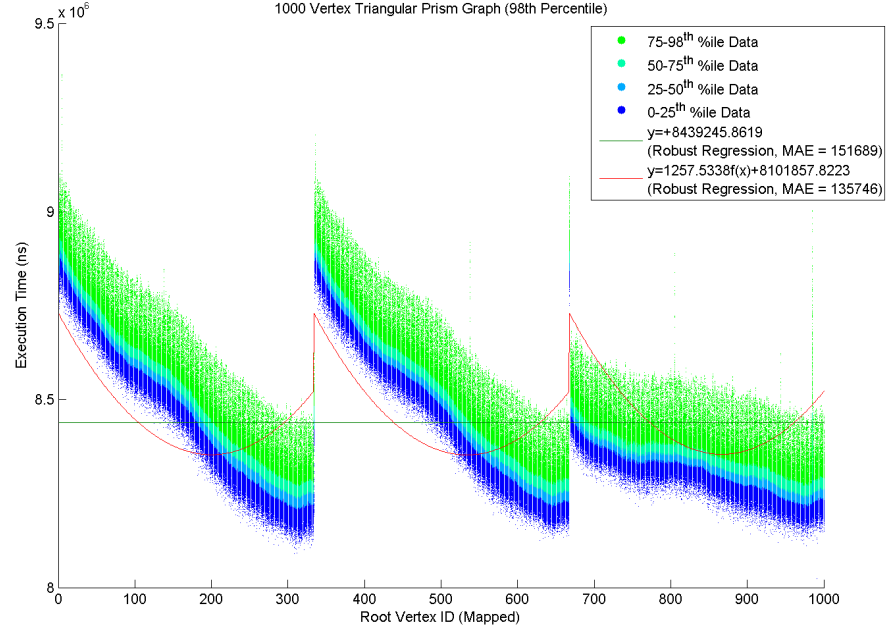


Figure 7.4.2: Execution Times for Varying Root Vertices of 1000 Vertex Triangular Prism Graph (98th%ile Data, Mapped Root ID)

Figure 7.4.2 shows the same data where:

- The vertex id of the root has undergone this mapping, clearly displaying the difference between the two curves; and
- The data above the 98th percentile has been removed for each vertex id; this eliminates the outlying higher execution time data points which significantly reduces the range of execution times being displayed and allows the lower quartiles to be more easily visualised.

Two best-fit curves were also investigated: a constant best-fit; and a best-fit of the mean depth of the segments within the segment tree (shown in dark green and red, respectively, in figure 7.4.2). These two best-fits were chosen as they allow a simple comparison between whether the execution time of the algorithm is affected or unaffected by variations in the structure of the segment tree as the root vertex changes.

The fits were calculated using iteratively re-weighted least squares and the constant execution time fit has Mean Absolute Error (MAE) of 151689 whereas the mean segment depth fit has MAE of 135746. The lower MAE indicates a better fit to the data, suggesting that changes to the composition of the segments and the structure of the segment tree influences the execution time. However, visual inspection shows that while the mean segment depth fit follows the data when the root vertex is closer to the centre (towards the left of each of the three partitions in figure 7.4.2) as the root vertex reaches $\sim 2/3$ of the way towards the outer triangle of vertices the mean segment depth starts to rise, diverging from the data, indicating that it does not fully model the execution time of the algorithm as the root vertex varies.

Expanding this to look at how varying the root vertex for Triangular Prism Graphs of different sizes affects execution time: appendix A contains figures A.1.1 – A.1.5 which show the empirical testing results for varying root vertex of (respectively) 250, 500, 750, 1250 and 1500 vertex Triangular Prism Graphs. These graphs show:

- Each graph shows a similar pattern of after mapping the vertex id such that the partitions of vertex ids show two approximately identical higher curves and one lower curve;
- The lower curve is in the middle partition of the vertex ids when modulus 3 of the number of vertices is zero (when there is no central vertex) and is in the right partition otherwise (when there is a central vertex);
- The shape of the mean segment-depth best-fit curve is similar as the graph size varies suggesting that the structure of the segment tree may be similar as the graph size increases; and
- Table 7.4.1 shows the MAEs for both mean segment-depth and a constant best-fit and, in all cases, the lower MAE (highlighted in green) is for the mean segment-depth best-fit.

Number of Vertices	MAE for Constant Fit	MAE for Mean Segment Depth Fit	Figure
250	19615	16692	A.1.1
500	71988	54032	A.1.2
750	118765	100171	A.1.3
1000	151689	135746	7.4.2
1250	196180	180767	A.1.4
1500	240885	221861	A.1.5

Table 7.4.1: Mean Absolute Error for Best-Fit Curves to Triangular Prism Results with Varying Root Vertices

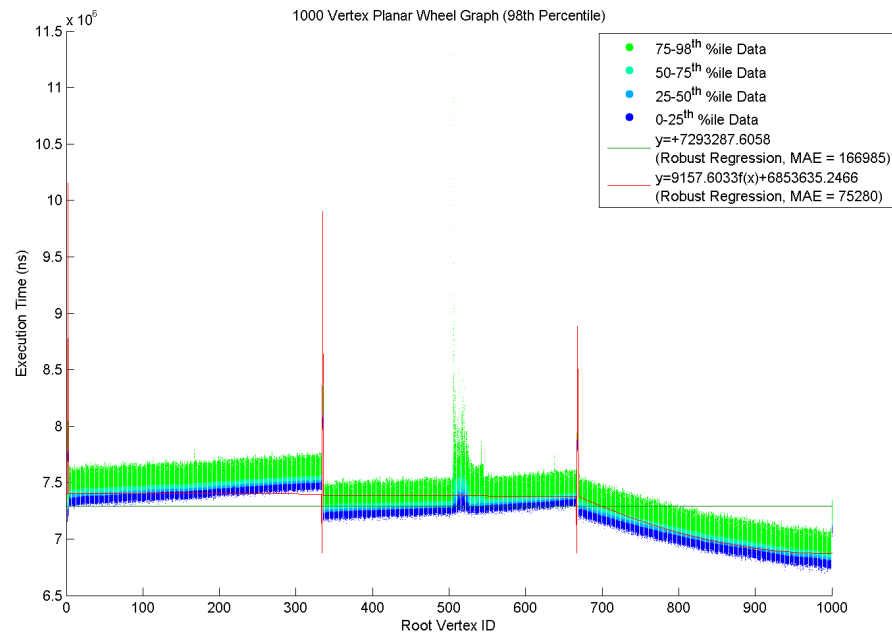


Figure 7.4.3: Execution Times for Varying Root Vertices of 1000 Vertex Planar Wheel Graph

Figure 7.4.3 shows the empirical testing results for a 1000 vertex Planar Wheel graph (unmapped and showing the 98th percentile results so the upper quartile does not excessively dominate the range of results). The plot shows these properties:

- Similar to the results for Triangular Prism graphs, the plot shows three groups of results corresponding to each of the three radial lines of vertices emanating from the centre of the wheel graph.
- The left-most two groups both show an approximately linear increase in execution time as the root vertex is further away from the centre.
- The right-most group shows a non-linear decrease as the root vertex moves further along that radial line of vertices away from the centre.
- The execution times for the hubs of the three wheels (at vertices 2, 335, & 668) show a marked increase compared to the results for the radial lines of vertices (although this is partly obscured by the mean segment-depth best-fit).
- There is an unusual spray of results for vertices with ids between 500 and 550 – after repeating the execution of this test this spray disappears (although other similar anomalies appear in other locations) suggesting that it is the impact of an external event on the benchmark and not a factor of the graph structure.
- The mean segment-depth best-fit for the left-most two groups of vertices is almost horizontal but has a slight downward trend, in contrast to the empirical results. Additionally, the outer-most vertex in each of these groups shows a drop in the mean segment-depth compared to the relatively constant values within the rest of the group of vertices and this drop is not present in the empirical testing data.
- The mean segment-depth best-fit for the right-most group of vertices shows a similar downward trend to the empirical data.
- The mean segment-depth best-fit at the hubs of the three wheels (at vertices 2, 335, & 668) show spikes in the depth of the segment tree that correspond to the increased execution time present in the empirical testing results.

Appendix A contains figures A.1.6 – A.1.10 that show the empirical testing results for 250, 500, 750, 1250 and 1500 vertex Planar Wheel Graphs that display similar features in both the execution time and for mean segment-depth. Table 7.4.2, below, lists the MAE between each best-fit curve and the empirical data and, in all cases, shows that the lower (highlighted in green) value corresponds to the mean segment-depth indicating that it provides a better fit to the data than a constant value.

Number of Vertices	MAE for Constant Fit	MAE for Mean Segment Depth Fit	Figure
250	35167	16282	A.1.6
500	89653	38959	A.1.7
750	97213	51124	A.1.8
1000	166985	75280	7.4.3
1250	176922	75365	A.1.9
1500	224613	80791	A.1.10

Table 7.4.2: Mean Absolute Error for Best-Fit Curves to Planar Wheel Results with Varying Root Vertices

Figure 7.4.4, below, shows the empirical testing results for a 1000 vertex Ordered Face Trisection graph (unmapped and showing the 98th percentile results so the upper quartile does not excessively dominate the range of results). Unlike the previous two graphs there is not a distinct pattern to either the empirical testing results nor the mean segment-depth as both are relatively constant across the domain of vertex ids; however, the mean segment-depth best-fit does have a marginally lower MAE when compared to the constant best-fit.

Appendix A contains figures A.1.11 – A.1.16 showing empirical testing results for 250, 500, 750, 1250, 1500 and 2000 vertex Ordered Face Trisection Graphs which display similar features to the 1000 vertex graph in both the execution time and for mean segment-depth. Table 7.4.3, below, lists the MAE between each best-fit curve and the empirical data and, in all cases, shows that the lower (highlighted in green) value corresponds to the mean segment-depth indicating that it provides a better fit to the data than a constant value; however, as noted for the 1000 vertex graph, this improvement is marginal in all cases.

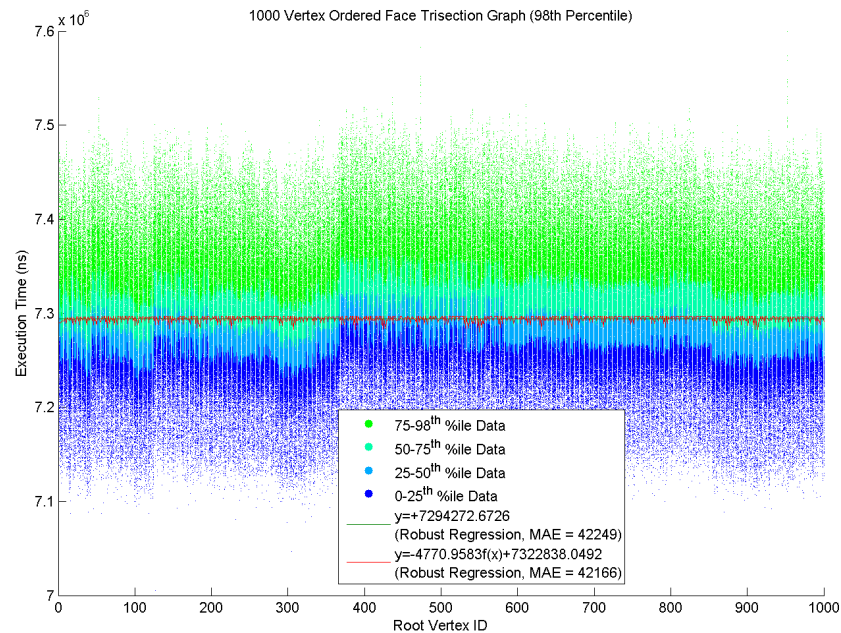


Figure 7.4.4: Execution Times for Varying Root Vertices of 1000 Vertex Ordered Face Trisection Graph

Number of Vertices	MAE for Constant Fit	MAE for Mean Segment Depth Fit	Figure
250	14947	14944	A.1.11
500	78655	78646	A.1.12
750	44280	44077	A.1.13
1000	42249	42166	7.4.4
1250	46152	46049	A.1.14
1500	140520	140420	A.1.15
2000	281160	281100	A.1.16

Table 7.4.3: Mean Absolute Error for Best-Fit Curves to Ordered Face Trisection Results with Varying Root Vertices

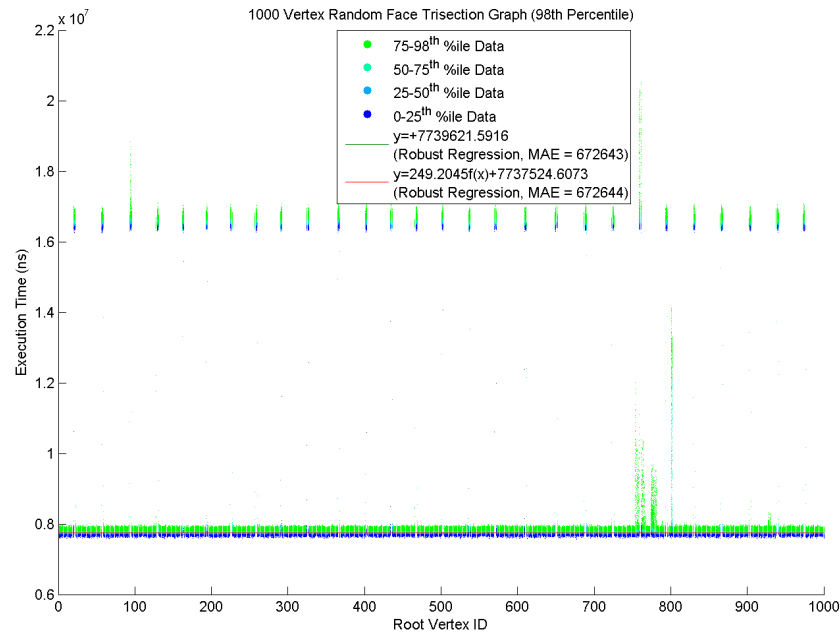


Figure 7.4.5: Execution Times for Varying Root Vertices of 1000 Vertex Random Face Trisection Graph

Figure 7.4.5 shows the empirical testing results for a 1000 vertex Random Face Trisection graph (unmapped and showing the 98th percentile results so the upper quartile does not excessively dominate the range of results). Similar to Ordered Face Trisection graphs, there is not a distinct pattern to either the empirical testing results nor the mean segment-depth as both are relatively constant across the domain of vertex ids; however, in contrast to the previous cases, the constant best-fit has a the lower MAE compared to the mean segment-depth best-fit but the difference is minimal.

The plot shows a series of increases (of a factor of ~2.1) in the execution time at approximately regular intervals. Repeating the benchmark with a reduced number of repetitions (Figure 7.4.6 – where the second result-set is displayed in blue-green overlayed with the previous result-set in black-red), on page 149, also shows a pattern of outlying results, however, the outliers are not in the same locations suggesting that these increases are not related to the structure of the graph or the implementation of the algorithm but instead due to an external factor.

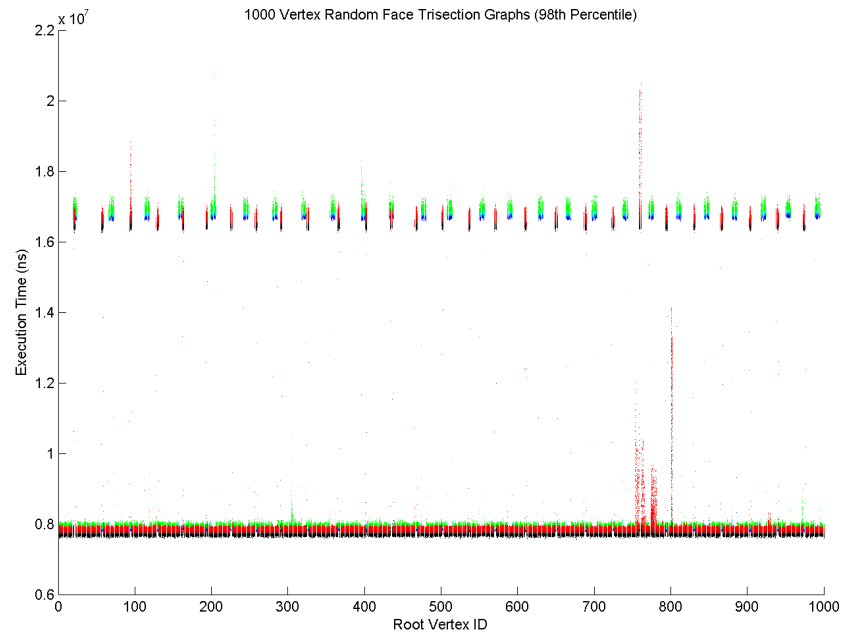


Figure 7.4.6: Two Overlaid Result-Sets for Execution Times for Varying Root Vertices of the Same 1000 Vertex Random Face Trisection Graph

Figure 7.4.7, below, shows the same data as in figure 7.4.5 with the execution time cut off at 8.1ms (removing the increased results from the visualisation; still showing 98th percentile data) and this highlights that the execution time as the root vertex is varied is approximately constant. Using this thresholded data, the mean segment-depth has a lower MAE than a constant best-fit but there is still no significant difference between the two fits.

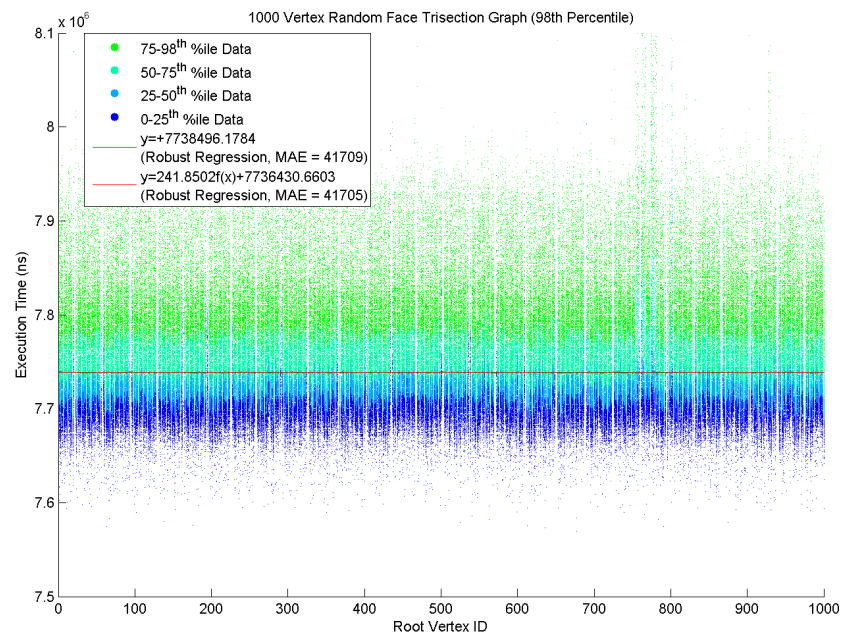


Figure 7.4.7: Execution Times for Varying Root Vertices of 1000 Vertex Random Face Trisection Graph (Results for Execution Times < 8.1ms)

Appendix A contains figures A.1.17 – A.1.21 showing empirical testing results for 250, 500, 750, 1250 and 1500 vertex Random Face Trisection Graphs which display similar features, to the 1000 vertex graph in figure 7.4.5, in both the execution time and for mean segment-depth. These graphs all show outlying periods of increased execution time as the root vertex varies:

- The outlying results are, for all graph sizes, a factor of, approximately, 2.1 compared to the majority of the result; and
- These periods are approximately regular. However, in the case of the 250 vertex graph they are not present initially.

Table 7.4.4, below, lists the MAE between each best-fit curve and the empirical data and shows that, in all cases, the fits have approximately the same error (which is unsurprising since the mean segment depth is approximately constant for all root vertices); however, with the exception of the 1000 vertex graph, there is a marginal trend towards the mean segment depth providing a better fit than a constant value.

Number of Vertices	MAE for Constant Fit	MAE for Mean Segment Depth Fit	Figure
250	100973	100968	A.1.17
500	262696	262693	A.1.18
750	476618	476605	A.1.19
1000	672643	672644	7.4.5
1250	773620	773615	A.1.20
1500	1022384	1022313	A.1.21

Table 7.4.4: Mean Absolute Error for Best-Fit Curves to Random Face Trisection Results with Varying Root Vertices

7.4.1 Conclusions for Varying Root Vertex for a Constant Graph Size and Structure

Two main conclusions can be drawn, regarding the mean segment depth, from the empirical testing of varying the root vertex used in the planarity testing whilst keeping the graph constant.

1. The results indicate that the mean segment depth typically gives a better (or at least as good as an) approximation, when compared to a constant value, when considering the variation in execution time with root vertex. However there is not a direct mapping from mean segment depth to execution time; as is evidenced by the triangular prism (figure 7.4.2) and planar wheel (figure 7.4.3) graph types where the mean segment depth demonstrates some attributes of the corresponding execution time results but there are also attributes which are not present.
2. The shape of the curves generated by the empirical testing and for the mean segment depth is similar as the graph size increases. Thus if a graph exhibits a characteristic at a given root vertex id (as assigned during generation) then as the graph size changes that same characteristic is present at a root vertex id which is proportional to the change in graph size.

Both of these suggest that graphs of the same type have similar structures to their segment hierarchies as their size increases.

The results also indicates that as a graph's size increases, for a given graph type, then the execution time of the planarity testing algorithm also increases. This result is explored further in subsequent sections of this chapter.

7.5 Empirical Testing on Maximal-Planar Graphs of Varying Size

This section considers the effect on execution time of the planarity testing algorithm (testing for planarity and generating a cyclic edge order) when, for maximal planar graphs of a similar structure, the graph's size is increased and the root vertex is constant (proportional to the graph size). The types graphs considered are: triangular prism; planar wheel; ordered face trisection; and random face trisection – representing maximal planar graphs with different properties.

Figure 7.5.1, below, shows the results of (up to) 1000 repetitions⁷ for each triangular prism graph with size from 42 vertices up to 4500 vertices, in increments of 3 vertices, where the root vertex is the graph's first (central) vertex. Four curves are overlaid on the graph showing the best-fits from the empirical testing results to $y = ax + b$ (red), $y = ax \cdot \ln(x) + b$ (green), $y = ax \cdot \ln(\ln(x)) + b$ (blue) and to the mean segment depth (purple – shown as $y = a \cdot f(x) + b$ in the legend); these curves overlap in the figure so the pertinent information relating to the curve is included in the legend, including the MAE (where the lowest MAE indicates the closest fit to the data).

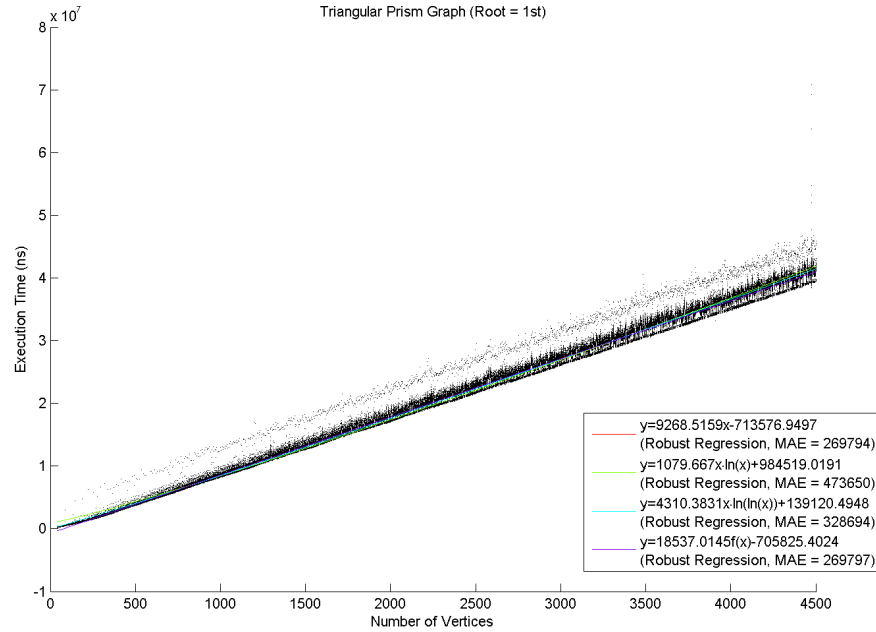


Figure 7.5.1: Triangular Prism Graph (42-4500 Vertices, 1st Vertex Root)

These best-fit curves show several properties:

- The lowest MAE is for a linear best-fit; however the mean segment depth has an almost identical MAE (which is unsurprising since, in this case, the mean segment depth is almost linear).
- The y -intercept of the linear best-fit is negative and the results for small graph sizes diverges from the linear fit to tend towards an execution time of zero (rather than an impossible negative execution time). It should be noted that a similar result was achieved for Hopcroft and Tarjan's algorithm where their test results indicated an execution time of $T = .0125V - .07$ [32, Pg. 565].

⁷ Repetitions interrupted by garbage collection are excluded from the results.

Table 7.5.1 shows the MAEs for the best-fits for empirical testing on graphs where, for a graph with N vertices, the root vertex is the 1^{st} , $1/4N^{\text{th}}$, $1/2N^{\text{th}}$, $3/4N^{\text{th}}$ and N^{th} vertex. The figures associated with these results are included in Appendix A.2.1 (with the exception of figure 7.5.1 which is included above). All these results are comparable to results when the root vertex is at the minimum such that:

- The lowest MAE for each results-set is for a linear best-fit (with the exception when the root vertex is at $1/2$ of the graph size, in which case, the lowest MAE is for the mean segment depth); however, in all cases, the mean segment depth and linear curves have almost identical MAEs (since the mean segment depth is approximately linear).
- The y -intercept of the linear best-fit for all these results-sets is negative.

The results-sets where the root vertex is the:

- $1/4N^{\text{th}}$ vertex shows a small “blip” in the results for graphs of small size (42-200 vertices) but is otherwise linear.
- $1/2N^{\text{th}}$ vertex shows a series of outlying results occurring at approximately regular time intervals (which appears at decreasing graph size intervals) where the outliers demonstrate an increase of a constant factor compared to the linear best-fit.
- N^{th} vertex shows a similar series of outlying results; however, the outliers initially occur regularly and then at greater intervals until they disappear.

Subsequent benchmarks (table 7.5.2, below) show some of the same combinations of graph type, size and root vertex but show different patterns of outliers (also a constant factor above the linear best-fit), within the same range of graph sizes, suggesting that these occurrences are indicative of an intermittent external event interrupting (or sharing a processor with) the planarity test rather than being symptomatic of the graphs or the algorithm.

Root	$y = ax + b$	$y = ax \ln(x) + b$	$y = ax \ln(\ln(x)) + b$	Mean Seg. Depth	Figure
1	269794	473650	328694	269797	7.5.1
$1/4N$	266796	459750	321899	266797	A.2.1
$1/2N$	2667103	2856678	2710051	2667039	A.2.2
$3/4N$	229296	437942	287909	229311	A.2.3
N	438760	645159	495399	438763	A.2.4

Table 7.5.1: Mean Absolute Error for Best-Fit Curves to Triangular Prism (42-4500 Vertices) Results

Table 7.5.2 shows the MAEs for the Triangular Prism graphs with the same root vertices as above but with graph sizes of 500-15000 vertices. Similar to above, the MAE indicates that a linear best-fit gives the best approximation to the results-set for each case of root vertex. Also, these results all have intermittent outliers which appear at approximately regular time intervals; as already noted above, these outliers appear for different graph sizes to the 42-4500 vertex benchmarks which implies that the outliers are not related to the graphs or the algorithm.

Root	$y = ax + b$	$y = ax \ln(x) + b$	$y = ax \ln(\ln(x)) + b$	Mean Seg. Depth	Figure
1	5953757	6407003	6074526	5953757	A.2.5
$1/4N$	10828822	11378716	11004559	10828827	A.2.6
$1/2N$	11151957	11380733	11176692	11151956	A.2.7
$3/4N$	11271872	11670634	11346354	11271872	A.2.8
N	7837889	8233416	7922696	7837889	A.2.9

Table 7.5.2: Mean Absolute Error for Best-Fit Curves to Triangular Prism (500-15000 Vertices) Results

Figure 7.5.2, below, shows the results of (up to) 1000 repetitions (results interrupted by garbage collection are excluded) for each planar wheel graph with size from 42 vertices up to 4500 vertices, in increments of 3 vertices, where the root vertex is the graph's first (central) vertex. Again, four curves are overlaid on the graph showing the best-fits from the empirical testing results to $y = ax + b$ (red), $y = ax \cdot \ln(x) + b$ (green), $y = ax \cdot \ln(\ln(x)) + b$ (blue) and to the mean segment depth (purple – shown as $y = a \cdot f(x) + b$ in the legend).

Several properties can be noted:

- Similar to the results for Triangular Prism graphs, the results show a series of intermittent outliers; these appear part way through the results-set (from graphs of ~ 2700 vertices) and from then occur at approximately regular time intervals;
- The linear and mean segment depth best-fit curves are almost identical and, correspondingly, have similar MAEs; and
- The MAEs for the best-fit curves indicates that $y = ax \cdot \ln(x) + b$ provides the best approximation to this results-set.

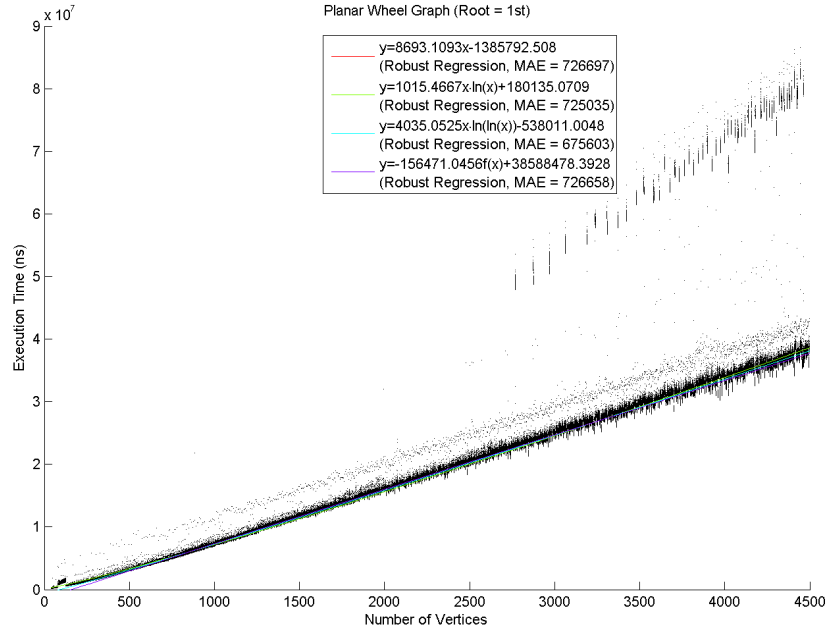


Figure 7.5.2: Planar Wheel Graph (42-4500 Vertices, 1st Vertex Root)

Table 7.5.3, below, shows the MAEs for that figure and for the results-sets of other root vertices for Planar Wheel graphs of 42-4500 vertices.

Root	$y = ax + b$	$y = ax \ln(x) + b$	$y = ax \ln(\ln(x)) + b$	Mean Seg. Depth	Figure
1	726697	725035	675603	726658	7.5.2
$1/4N$	906099	931009	859604	906223	A.2.10
$1/2N$	3712651	3776957	3694559	3712739	A.2.11
$3/4N$	599781	613808	552187	599849	A.2.12
$3/4N$ (2)	3968010	3919066	3921366	3967942	A.2.13
N	301621	330432	254431	8520598	A.2.14

Table 7.5.3: Mean Absolute Error for Best-Fit Curves to Planar Wheel (42-4500 Vertices) Results

In all cases, these results show:

- The best approximation to the curve is not given by a linear best-fit.
- There are outlying results which are a constant factor above the main body of results.

Examining the reason for the apparent non-linearity, table 7.5.4 shows the same result-sets with the tiny graph sizes excluded from the analysis. In these cases, the MAEs all indicate that the results-sets are best approximated by a linear fit and it is only the inclusion of graphs with a small (≤ 750) number of vertices which distorts these results. As previously noted for Triangular Prism graphs, the y -intercept of the linear best-fit is negative (as is the case here); thus the linear trend cannot be followed for these small graph sizes else they would have a negative (impossible) execution time.

Root	Minimum Graph Size	$y = ax + b$	$y = ax \ln(x) + b$	$y = ax \ln(\ln(x)) + b$	Figure
1	750	761872	805531	767938	A.2.15
$1/4N$	750	1000452	1034677	1001089	A.2.16
$1/2N$	500	4055350	4149058	4079101	A.2.17
$3/4N$	750	603453	644836	606917	A.2.18
$3/4N$ (2)	750	4517559	4548268	4520036	A.2.19
N	750	274585	315838	277078	A.2.20

Table 7.5.4: Mean Absolute Error for Best-Fit Curves to Planar Wheel (Up to 4500 Vertices) Results After Removal of Tiny Graph Sizes

One additional point that can be made about these results. Table 7.5.3 shows the MAEs for the results-sets for different root vertices and in all cases, except for graphs of N vertices when the root vertex is the N^{th} vertex, the MAE for the linear fit and the mean segment depth fit are approximately equal (since the mean segment depth is approximately linear). In the exception, the mean segment depth is not linear; instead (as noted in section 7.3.8, on page 138) as the graph size increases the mean segment depth tends towards $23/6$ and the mean segment depth best-fit has an MAE an order of magnitude above the other best-fits.

Therefore, where mean segment depth has provided some limited insight into how the execution time changes as the graph size is constant and the root vertex varies – this does not follow into the case where the graph size is varied and the root vertex is constant (proportional to the graph size). It has only appeared that way for the previous results as, coincidentally, the mean segment depth has increased approximately linearly with graph size.

Root	$y = ax + b$	$y = ax \ln(x) + b$	$y = ax \ln(\ln(x)) + b$	Mean Seg. Depth	Figure
1	1238930	1723401	1385571	1238934	A.2.21
$1/4N$	14849883	15158460	14890372	14849903	A.2.22
$1/2N$	14165175	14614205	14302030	14165170	A.2.23
$3/4N$	11814445	12225130	11941998	11814445	A.2.24
N	7038255	7545315	7219387	26373698	A.2.25

Table 7.5.5: Mean Absolute Error for Best-Fit Curves to Planar Wheel (500-15000 Vertices) Results

Table 7.5.5 shows the MAEs for the same root vertices of 500-15000 vertex Planar Wheel graphs. In all cases, the MAEs indicate that the best approximation to the results-sets is given by a linear fit and or by the mean segment depth fit when it is, correspondingly, linear.

It can also be noted, that each of those results-sets shows outliers to a greater ($1/2N^{\text{th}}$ root)

or lesser (1st root) degree but at different points to the previous results sets for 42-4500 vertex Planar Wheel graphs.

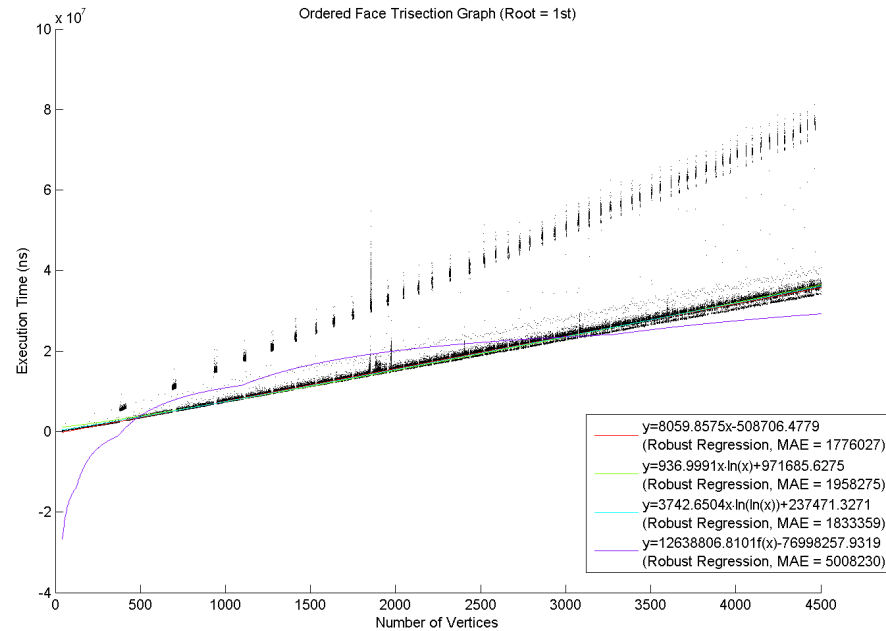


Figure 7.5.3: Ordered Face Trisection Graph (42-4500 Vertices, 1st Vertex Root)

Figure 7.5.3, below, shows the results for Ordered Face Trisection graphs of 42-4500 vertices. It shows the same properties as the previous graph types:

- The best approximation to the data is a linear best-fit; and
- Intermittent outliers are present in the results-set at constant time intervals.

However, as per the Planar Wheel graph when the root vertex is the N^{th} vertex, it also has a non-linear progression to the mean segment depth and, correspondingly, the mean segment depth provides the worst fit to the data.

Table 7.5.6, below, shows a summary of the MAEs from that figure and from the results of the other root vertices. All these results-sets have the same properties; however, where the mean segment depth for the first vertex is a smooth, if bumpy, curve the other root vertices follow a similar curve but it is not smooth as there is much more noise (spikes in the best-fit line) above this curve.

Root	$y = ax + b$	$y = ax \ln(x) + b$	$y = ax \ln(\ln(x)) + b$	Mean Seg. Depth	Figure
1	1776027	1958275	1833359	5008230	7.5.3
$1/4N$	3665093	3854375	3722815	8571908	A.2.26
$1/2N$	1225056	1414951	1285655	5953908	A.2.27
$3/4N$	1553096	1725039	1600865	7000747	A.2.28
N	1406122	1598605	1467602	5889114	A.2.29

Table 7.5.6: Mean Absolute Error for Best-Fit Curves to Ordered Face Trisection (42-4500 Vertices) Results

Table 7.5.7, below, shows the results for the same root vertices for Ordered Face Trisection graphs with 500-15000 vertices and indicates, again, that: the best approximation to the data is given by a linear fit; and that the mean segment depth fit is the worst approximation to the data (having approximately twice the MAE of a linear fit).

Root	$y = ax + b$	$y = ax \ln(x) + b$	$y = ax \ln(\ln(x)) + b$	Mean Seg. Depth	Figure
1	6278093	6659470	6351021	14929242	A.2.30
$1/4N$	11120304	11551287	11233880	23502400	A.2.31
$1/2N$	12676386	13071826	12785937	28158443	A.2.32
$3/4N$	12166627	12545700	12280494	23864311	A.2.33
N	10363658	10799158	10480651	24069355	A.2.34

Table 7.5.7: Mean Absolute Error for Best-Fit Curves to Ordered Face Trisection (500-15000 Vertices) Results

Figure 7.5.4, below, shows the results for 42-4500 vertex Random Face Trisection graphs where the root vertex is the 1st (central) vertex. Again, showing: that the best approximation to the data is a linear fit; that the mean segment depth is the worst approximation to the data with an MAE of approximately twice that for the other fits; and there are intermittent outlying results which regular time intervals. The result-set also shows a similar pattern in the mean segment depth as for Planar Wheel and Ordered Face Trisection graphs of the same, 1st, root vertex; in these cases the mean segment depth is non-linear and asymptotic.

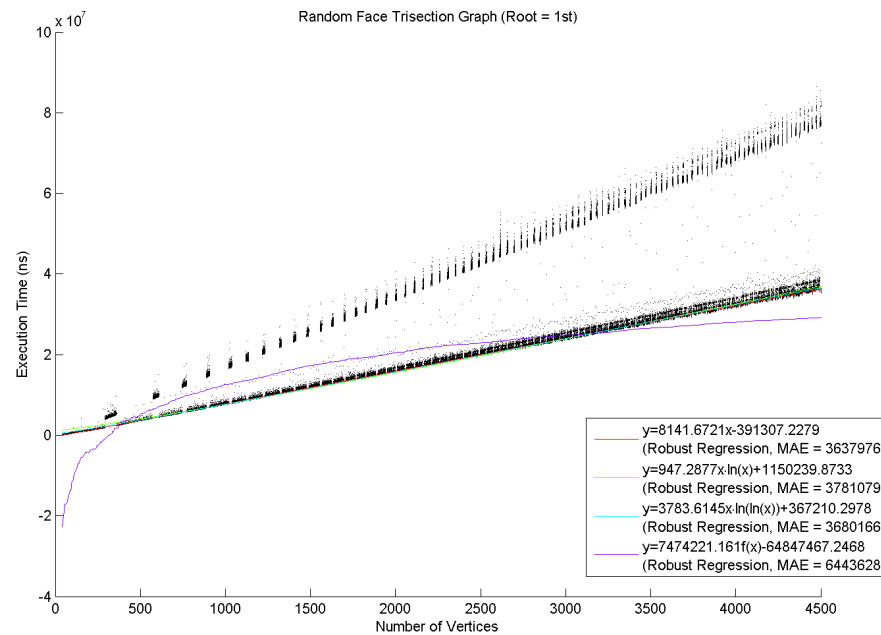


Figure 7.5.4: Random Face Trisection Graph (42-4500 Vertices, 1st Vertex Root)

Table 7.5.8 summarises the MAE for that results-set and for the results from other root vertices for Random Face Trisection graphs of 42-4500 vertices. In all cases:

- The linear best-fit has a negative y -intercept and the data diverges from the linear best-fit when the graph sizes are small.
- The mean segment depth best-fit has the highest MAE, indicating that it is the worst approximation to the data. Also, all the mean segment depths appear asymptotic as the graph size increases; however when the root vertex is not the 1st vertex the mean segment depth follows a similar curve to the smooth curve for that 1st vertex root although there is a lot of noise around this curve.
- There are intermittent outliers in the results which appear at approximately regular time intervals.

In most cases, the linear best-fit has the lowest MAE, indicating that it is the best approximation to the data. In the case where the root vertex is the $3/4N^{\text{th}}$ vertex the log-log best-fit is the best approximation; however, repeating the analysis with same data but excluding graphs containing less than 400 vertices then linear best-fit gives the best approximation – indicating that the small graph sizes skewed the analysis in this case.

Root	Minimum Graph Size	$y = ax + b$	$y = ax \ln(x)$ +b	$y = ax \ln(\ln(x))$ +b	Mean Seg. Depth	Figure
1	42	3637976	3781079	3680166	6443628	7.5.4
$1/4N$	42	2116531	2293644	2172621	6335956	A.2.35
$1/2N$	42	2410211	2588612	2467059	7212265	A.2.36
$3/4N$	42	2632114	2679508	2618944	7039375	A.2.37
$3/4N$	400	2759947	2854157	2782140	–	A.2.39
N	42	1816324	1953147	1853941	6737186	A.2.38

Table 7.5.8: Mean Absolute Error for Best-Fit Curves to Random Face Trisection (up to 4500 Vertices) Results

Table 7.5.9 shows similar results for the same (proportionally) root vertices for Random Face Trisection graphs of up to 15000 vertices. These results show the same properties as above.

Root	Minimum Graph Size	$y = ax + b$	$y = ax \ln(x)$ +b	$y = ax \ln(\ln(x))$ +b	Mean Seg. Depth	Figure
1	500	8992789	9277734	9021704	17181059	A.2.40
$1/4N$	500	7586314	7909207	7629515	20201626	A.2.41
$1/2N$	500	10022019	10332961	10064013	25609920	A.2.42
$3/4N$	500	9945233	10076222	9935005	27912475	A.2.43
$3/4N$	750	10056748	10270388	10075262	–	A.2.45
N	500	8474118	8778526	8523768	22573254	A.2.44

Table 7.5.9: Mean Absolute Error for Best-Fit Curves to Random Face Trisection (up to 15000 Vertices) Results

7.5.1 Conclusions for Empirical Testing on Maximal-Planar Graphs of Varying Size

Several conclusions can be drawn from the results in this section:

1. The best approximation is a linear best-fit between the algorithm's execution time and the graph size for a fixed (maximal planar) graph type and a fixed (as a proportion of the size of the graph) root vertex;
2. The mean segment depth does not give a good approximation to the algorithm's execution time, unless it is also (approximately) linear;
3. The linear best-fit has a negative y -intercept (as is the case for Hopcroft and Tarjan's algorithm [32, Pg. 565]);
4. Expanding the previous point – for graphs with small (≤ 750) numbers of vertices, the empirical testing results diverged from the, corresponding, linear-best fits (which, given the negative y -intercept, predicts near zero or negative execution times) and these small graphs could distort the best-fit analysis emphasising a non-linear fit where that fit was less appropriate when considering the graphs larger than that limit; and
5. The results were regularly and repeatedly interrupted by an intermittent external event which adds a constant scale factor to those results generated in within its time frame.

7.6 Empirical Testing on Graphs of Constant Vertex-Size and Varying Edge-Size

This section extends the previous to look at biconnected graphs, rather than just maximal planar graphs, and considered the case when multiple embeddings of a graph are possible.

Table 7.6.1, below, summarises the empirical testing results for biconnected sub-graphs (with between approximately 30,000 and 45,000 edges) generated by deleting cotree edges (which are each the sole edge of a segment) of 14999 vertex Triangular Prism, Planar Wheel, Ordered Face Trisection and Random Face Trisection graphs and for different root vertices. The table contains the MAEs for the best-fits $y = ax + b$, $y = ax \ln(x) + b$ and $y = ax \ln(\ln(x)) + b$ for each of these graph types and the lower the MAE the better the approximation that fit is towards the data.

Graph Type	Root	$y = ax + b$	$y = ax \ln(x) + b$	$y = ax \ln(\ln(x)) + b$	Figure
TP	1	15541986	15542722	15542137	A.3.1
	$1/4N$	14788546	14765916	14779177	A.3.2
	$1/4N$ (2)	19076118	19055515	19067521	A.3.3
	$1/2N$	14014165	14023381	14017703	A.3.4
	$3/4N$	13953239	13957438	13954722	A.3.5
	N	14156578	14160763	14158026	A.3.6
PW	1	16400791	16405925	16402689	A.3.7
	$1/4N$	17279038	17281242	17279758	A.3.8
	$1/2N$	13161375	13174988	13166686	A.3.9
	$3/4N$	12951990	12965026	12957054	A.3.10
	N	16315376	16316008	16315450	A.3.11
OFT	1	20678436	20679501	20678696	A.3.12
	$1/4N$	22395179	22392774	22394051	A.3.13
	$1/4N$ (2)	16173935	16181328	16176750	A.3.14
	$1/2N$	18556445	18557985	18556870	A.3.15
	$3/4N$	16568619	16574276	16570723	A.3.16
	N	20754869	20756816	20755480	A.3.17
RFT	1	16861982	16867804	16864137	A.3.18
	$1/4N$	16791247	16794352	16792280	A.3.19
	$1/2N$	14939222	14948367	14942765	A.3.20
	$3/4N$	15082308	15083301	15082542	A.3.21
	N	15997322	16002293	15999185	A.3.22

Table 7.6.1: Mean Absolute Error for Best-Fit Curves to Biconnected Triangular Prism (TP), Planar Wheel (PW), Ordered Face Trisection (OFT) and Random Face Trisection (RFT) Sub-Graph Results (14999 Vertices, ~30000-45000 Edges, Various Root Vertices)

The MAEs show that in most cases a linear fit provides the best approximation to the data. However there are two cases where this is not the case that are discussed below.

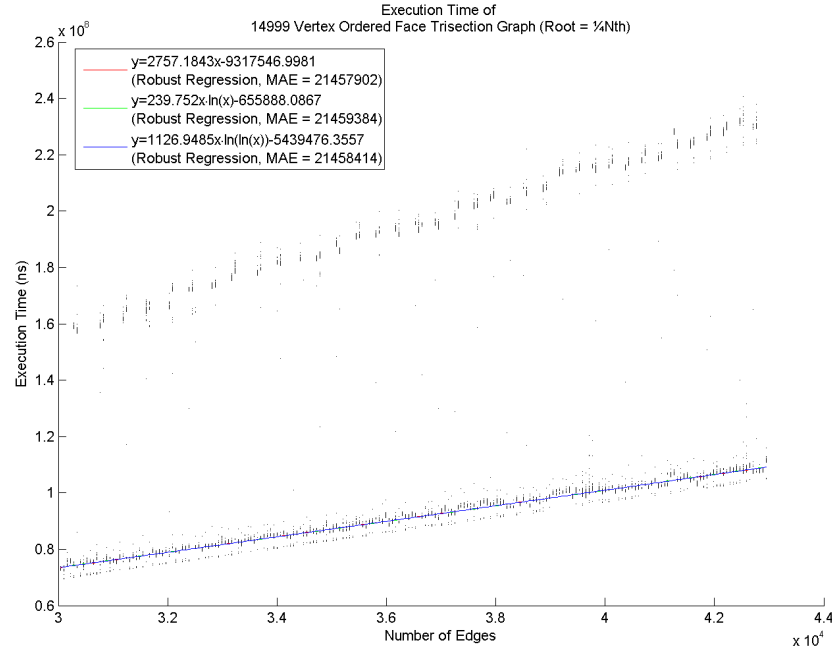


Figure 7.6.1: Ordered Face Trisection Graph (14999 Vertices, ~ 30000 - 43000 Edges, $1/4N^{\text{th}}$ Vertex Root)

The more trivial of the cases where a linear fit is not indicated by the MAEs is the Ordered Face Trisection sub-graphs when the root vertex is the $1/4N^{\text{th}}$ vertex. Examining figure A.3.13 (page 216) shows a small increase in execution times at about $44,000 \pm 500$ edges (and is more noticeable in the outliers) but this is enough to skew the MAEs towards a non-linear fit. Figure 7.6.1, above, shows the same data with the results for sub-graphs with 43,000-45,000 vertices excluded and the MAEs are summarised in table 7.6.2, below. With these exclusions the results indicate a linear fit is the best approximation to the data.

Performing a second run (figure A.3.14, page 216) of that benchmark (MAEs summarised in table 7.6.1, above) validates this as there is no similar increase in execution times within that edge range for that run and the MAEs indicate that, without any exclusions, a linear fit gives the best approximation to the data. This suggests that the increase in execution times in first benchmark was an anomaly.

Root	Edges	$y = ax + b$	$y = ax \ln(x) + b$	$y = ax \ln(\ln(x)) + b$	Figure
$1/4N$	≤ 43000	21357341	21359338	21358064	7.6.1

Table 7.6.2: Mean Absolute Error for Best-Fit Curves to Ordered Face Trisection Sub-Graph Results (14999 Vertices, Varying Edges)

Of more interest are the two benchmarks performed for Triangular Prism sub-graphs where the root vertex is the $1/4N^{\text{th}}$ vertex. Examining figures A.3.2 & A.3.3 (page 209) shows that a step is apparent in the results at approximately 40,000 edges.

Figures 7.6.2 & 7.6.3, below, show these results-sets split at 40,000 vertices and the MAEs are summarised in table 7.6.3; these results indicate that above and below this step the best approximation to the data is a linear fit.

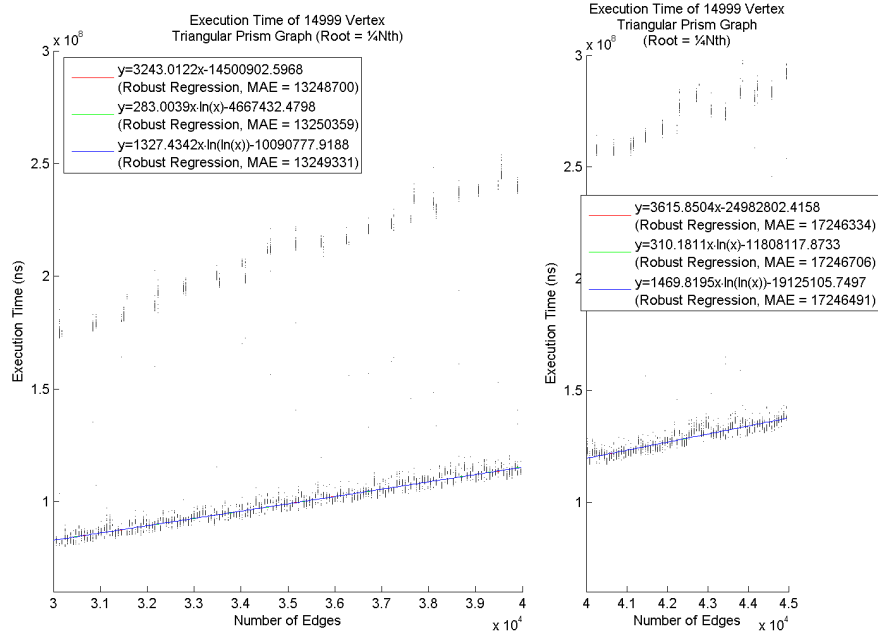


Figure 7.6.2: Biconnected Sub-Graphs of Triangular Prism Graph (14999 Vertices, ~30000-45000 Edges (Split at 40000 Edges), $\frac{1}{4}N^{\text{th}}$ Vertex Root)

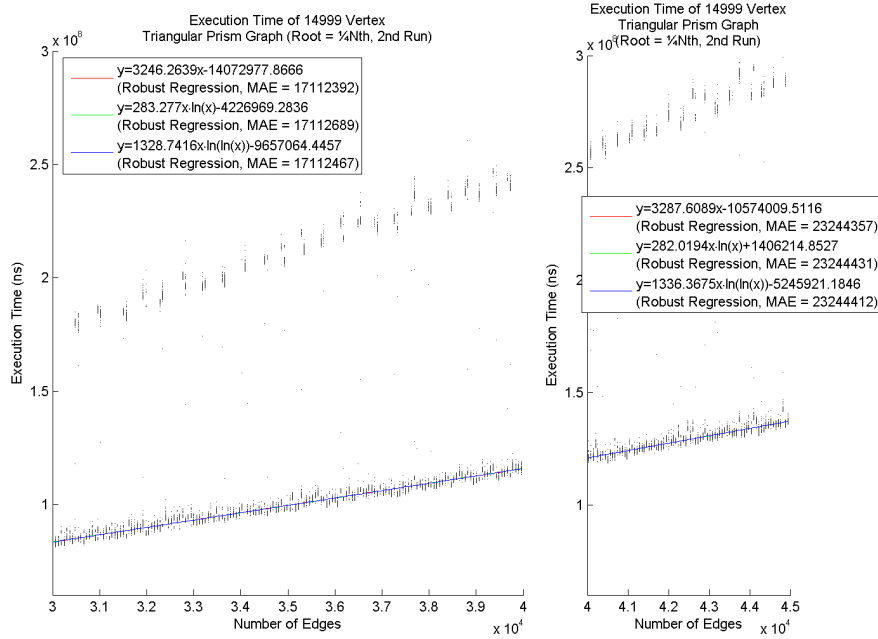


Figure 7.6.3: Biconnected Sub-Graphs of Triangular Prism Graph (14999 Vertices, ~30000-45000 Edges (Split at 40000 Edges), $\frac{1}{4}N^{\text{th}}$ Vertex Root)

Root	Edges	$y = ax + b$	$y = ax \ln(x) + b$	$y = ax \ln(\ln(x)) + b$	Figure
$\frac{1}{4}N$	≤ 40000	13248700	13250359	13249331	7.6.2
$\frac{1}{4}N$	≥ 40000	17246334	17246706	17246491	7.6.2
$\frac{1}{4}N(2)$	≤ 40000	17112392	17112689	17112467	7.6.2
$\frac{1}{4}N(2)$	≥ 40000	23244357	23244431	23244412	7.6.2

Table 7.6.3: Mean Absolute Error for Best-Fit Curves to Triangular Prism Sub-Graph Results (14999 Vertices, Varying Edges)

Figure 7.6.4 shows some of the changing properties relating to the structure of the Triangular Prism sub-graphs for that $1/4N^{\text{th}}$ root vertex. It shows two very noticeable traits:

1. The sub-graphs with less than 40,000 edges have no conflicts between segments whereas those with greater than 40,000 edges have an (approximately linearly) increasing number of conflicts between segments.
2. The mean segment depth for those sub-graphs shows a maxima at approximately 34,500 edges and a point of inflexion at 40,000 vertices; corresponding to this point of inflexion is a distinct change to the gradient of the mean segment depth curve and, even more distinct, the acceleration of the mean segment depth curve changes from a smooth curve to a pattern reminiscent of harmonic dampening.

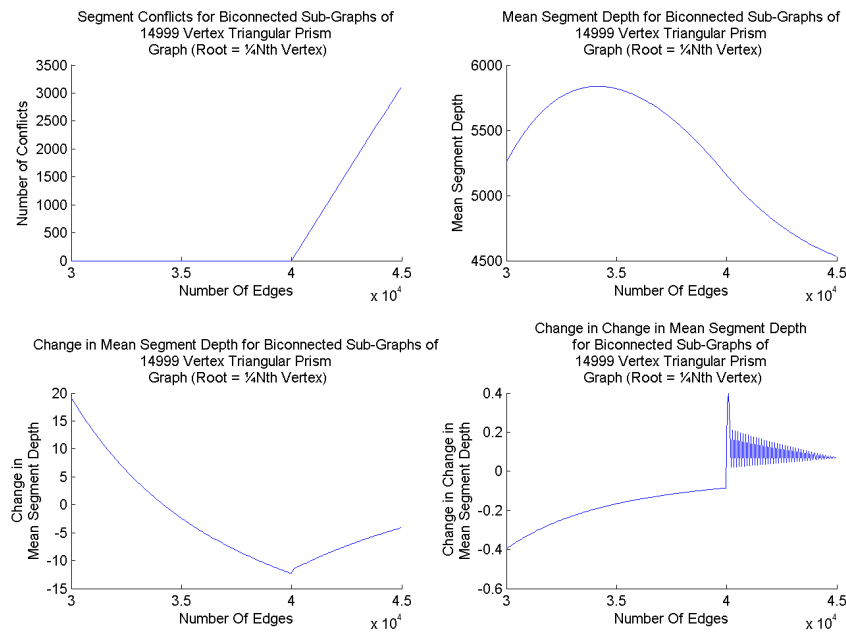


Figure 7.6.4: Mean Segment Depth for Biconnected Sub-Graphs of Triangular Prism Graph (14999 Vertices, ~ 30000 -45000 Edges, $1/4N^{\text{th}}$ Vertex Root)

Appendix A.4 shows the changes to the numbers of conflicts and to the mean segment depth for the other graph types and root vertices. The only other instance of a similar change is also for the Triangular Prism sub-graphs when the root vertex is the $1/2N^{\text{th}}$ vertex (figure A.4.2, page 222); this results-set shows a change in the number of conflicts and a point of inflexion in the mean segment depth curve (and, corresponding, change in the gradient and acceleration of the mean segment depth from a smooth curve to a reducing sine wave). Examining the empirical data for this results set (figure A.3.4, page 210) also shows a small step in the results but it is almost inconsequential⁸ compared to the step for the $1/4N^{\text{th}}$ root vertex and the linear fit is still the best approximation to the data in this case.

The other graph types and root vertices (figures contained in appendix A.4) shows comparable changes in gradient to the two above but none of the other graphs show such a distinct change in the acceleration of the mean segment depth curve from a smooth curve to a sinusoidal curve of reducing amplitude.

- The other Triangular Prism results-sets show either a smooth curve or a reducing amplitude

⁸ And probably would not have been remarked upon if the mean segment depth had not been analysed

sinusoidal curve but they do not both occur together outside the $1/4N^{\text{th}}$ and $1/2N^{\text{th}}$ root vertex results-sets.

- The Planar Wheel results-sets all show smooth curves for the mean segment depth with reducing gradient until an approximately constant gradient is reached.
- The Ordered Face Trisection results-sets all have a single change in gradient at about 40,000 edges for the number of conflicts and the mean segment depth is characterised by discrete steps in the gradient separated by ranges of near constant gradient.
- The Random Face Trisection result-sets all show a smooth curve for the number of conflicts and for the mean segment depth; although the gradient and the acceleration of the mean segment depth curve for this graph type show a lot of “random” noise, it is over a small range (which for the acceleration is consistently around zero except for when the number of edges approaches 45,000 when the acceleration increases negatively).

A much more detailed analysis of these effects is needed along with a method of mutating the graph structure to generate different biconnected graphs with similar properties if these relationships are to be explored further. However, the lack of steps in the results-sets of the other graph types seems to indicate that:

- The steps in the results-sets are not related to changes in the gradient of the frequency of conflicts as most results-sets show a sharp variation in the gradient and no step is present; and
- Changes in the curve/gradient/acceleration for the mean segment depth are insufficient to induce a step in the results-sets.

However, a fundamental change in the pattern of acceleration for the mean segment depth curve (as can be seen for those two Triangular Prism sub-graph results-sets but is not in evidence for the others) may be the cause (or a related symptom) of the existence of that step in the execution times.

7.7 Empirical Testing on Permutable, Flippable Graphs of Varying Size

Table 7.7.1, below, summarises the MAEs from the execution times of various root vertices for Permutable, Flippable graphs (see section 7.3.5, page 134) with 500-7500 facets (1002-15002 vertices).

Root	$y = ax + b$	$y = ax \ln(x) + b$	$y = ax \ln(\ln(x)) + b$	Mean Seg. Depth	Figure
1	8793283	8890456	8804118	20124000	A.5.1
$1/4N$	9588673	9777305	9642010	19954875	A.5.2
$1/2N$	10154894	10320925	10201495	20405195	A.5.3
$3/4N$	9775322	9943227	9822890	20167180	A.5.4
N	9766313	9882703	9796296	19937485	A.5.5

Table 7.7.1: Mean Absolute Error for Best-Fit Curves to Permutable Flippable Graph Results (1002-15002 Vertices)

The table shows that, in all cases, a linear fit gives the best approximation to the data and that the mean segment depth is the worst approximation to the data with approximately double the error of the other fits.

The results for Permutable, Flippable graphs when the root vertex is the $1/4N^{\text{th}}$, $1/2N^{\text{th}}$, $3/4N^{\text{th}}$ and N^{th} (figures A.5.2-A.5.5) all show a positive y -intercept; however, as with all previous results, when the root is the 1st vertex the y -intercept is negative.

7.8 Conclusions

Considering the variation in the root vertex:

- Varying the root vertex for Triangular Prism and Planar Wheel graphs varies the execution time such that and the results-sets for the same graph type exhibited similar trends in the execution time as root vertex varied as a proportion of the graph size.
- A similar property can be noted for Ordered Face Trisection and Random Face Trisection graphs; however this is to a lesser degree as the execution times are approximately constant as the root vertex varied.
- In all cases, the mean segment depth best-fit gave a better or approximately equal approximation to the execution time than a constant value; this was particularly noticeable for the Triangular Prism and Planar Wheel graphs where the execution time and mean segment depth showed some significant variation as the root vertex changed (whereas the Ordered Face Trisection and Random Face Trisection graphs were relatively constant in both aspects). However, while the mean segment depth and the execution time share some characteristic, the mean segment depth does not always model the behaviour of the execution time and so other factors may need to be identified and considered to achieve a better model.

Considering the variation in execution time for a corresponding variation in the graph size for maximal planar graphs of the same type and (proportional to the graph size) root vertex:

- Typically, the analysis showed that the best approximation to the results-sets was given by a linear best-fit.
- This linear best-fit took the form $y = ax - b$ with a negative y -intercept; this is a comparable result to Hopcroft and Tarjan who noted that their planarity testing algorithm achieved an execution time of $T = .0125V - .07$ [32, Pg. 565] (also exhibiting a negative y -intercept).
- Where the analysis did not indicate a best approximation by a linear best-fit then excluding results for small graphs of below 750 vertices indicated that, for the graphs larger than this size, a linear best-fit was the best approximation. This is a side-effect of finding a linear best-fit with a negative y -intercept (which means that the execution times for small graphs must diverge from the linear best-fit as negative execution times are impossible to achieve) and, since fitting curves tends to be more sensitive to anomalies and outliers at the extremes of the domain, this results in promoting a non-linear best-fit in these cases where it is not necessarily appropriate.
- The results-set typically contained outlying data which appeared intermittently at approximately constant time intervals but these intervals were independent of the root vertex, graph type or size (as they varied with repeated benchmarks of the same graph and root vertex combinations); so the conclusion drawn is that these outliers are the impact of an external event interrupting (or sharing the processor with) the planarity testing algorithm's execution and are otherwise irrelevant.

Considering the variation in execution time for a corresponding variation in the number of the graph's edges for biconnected sub-graphs of a fixed maximal planar graph and root vertex:

- In all cases, the analysis showed that the best approximation to the results-set was given by a linear best-fit.

- Again, this linear best-fit took the form $y = ax - b$ with a negative y -intercept.
- Where the change in gradient of the mean segment depth displayed a distinct variation from a smooth curve to a sinusoidal curve of reducing amplitude (as per the Triangular Prism graphs for the $1/4N^{\text{th}}$ and $1/2N^{\text{th}}$ root vertices) there was a step in the execution times at a corresponding number of edges and that the best approximation to the results-set above and below the step was a linear best-fit.

Similarly, the results for Permutable, Flippable graphs showed that a linear fit gave the best approximation to the data; however, contrary to the previous results, the y -intercept for this fit against the majority of the results-sets was positive.

Drawing this all together then the overall conclusion is that:

CONCLUSION 7.8.1. The empirical testing shows that, for graphs with more than 750 vertices, the best approximation to the data is a linear best-fit.

More specifically, since each phase of the algorithm is theorised to execute in linear time and when tested empirically the sum of the execution times for all phases (including generating a cyclic edge order) was found to be linear then this implies that each individual phase of the algorithm is linear (since if any phase was non-linear then the total execution time would likely have been non-linear).

Further Analysis

Further analysis can be performed to consider whether it is possible to generate a model that can approximate the behaviour of the algorithm:

1. The results have shown that changes in mean segment depth as the number of vertices or edges increases does not give a good approximation to the change in execution time; however, the mean segment depth did approximate (to some degree) changes in the execution time as the root vertex varied and the graph was otherwise fixed. Given this then one avenue of investigation would be whether the mean segment depth gives any approximation to the gradient of the linear fit against execution time as the number of vertices and edges increases.
2. Other factors (such as numbers of conflicts, blocks, children, etc.) can also be considered to see whether they can be used to model the behaviour of the algorithm. cursory analysis of the data rejected those models listed above and several others relating to the combination of segment depth multiplied by numbers of children or of conflicts; however, a more detailed analysis would be useful to validate this viewpoint.
3. For most graph type, size and root vertex combinations the linear best-fit's y -intercept is negative; however, for Flippable, Permutable graphs where large proportions of the segments being either part of a *permutable* group or of *flippable* blocks then the y -intercept was, for the most part, positive. Repeating these benchmarks and generating new benchmarks with graphs of similar properties would validate whether the number of *permutable* or *flippable* segments has any correlation to the y -intercept or if these results were coincidence.

One particular area where more analysis could be focused is on biconnected sub-graphs of triangular prism graphs – in particular, what causes the step to appear in the execution times and whether it is linked to any particular structure in the graph or the segment hierarchy.

Chapter 8

Conclusions

This chapter draw together the conclusions from the previous chapters:

Chapter 2 gives an overview of the field of planarity testing and the various approaches that have been taken to the problem. All the published algorithms which consider planarity testing and generate an embedding form a single embedding of that graph and it is not possible to manipulate them to generate alternate embeddings. Although, Haeupler and Tarjan published an extended abstract giving a brief summary of a method for generating all possible embeddings using PQ-trees, there is no detail to the algorithm making it difficult (impossible?) to implement. As such, the conclusion was drawn that there were no published algorithms to generate all embeddings of a biconnected graph in a time bound linear to the size of the graph (for each embedding).

Chapters 3-4 gives the theory behind the algorithm detailing how and why it works. The explanation of the algorithm is deeply linked to the properties of Trémaux Trees and the subsequent generation and ordering of Segments induced from this tree structure. The resulting planarity test stems from the invariant properties associated with these structures and orderings and these two chapters detail these invariant properties and the relationships needed to test for planarity and, if successful, perform an embedding.

Given these invariant properties and the relationships defined by the hierarchy and ordering of segments then the algorithm (chapter 5) is relatively simple as correct generation of: the Trémaux Tree; the segments; and their ordering induces all the invariant properties and then it becomes a matter of checking whether each segment and block is either fully embedded or whether they can be flipped so as to permit an embedding.

In the consideration of the theory, no restriction was placed on the nature of the graph beyond the fact that it should be connected. Chapters 3-4 consider the case of separable and biconnected graphs equally and show that no differentiation needs be made between the two for the purposes of planarity testing as the algorithm can test both equally well. Thus the conclusion is drawn that Hopcroft and Tarjan's restriction on biconnected graphs can be relaxed and any connected graph can be tested via a path addition method.

Chapter 5 also contains an algorithm to allow permutation of a biconnected graph via either Whitney Flips or permutation of edges with an equivalent precedence ordering so as to achieve all possible embeddings of a biconnected graph in linear time complexity. Chapter 4 also details techniques to generating all embeddings of Class 1 and 2 segments about each common head vertex in a linear time complexity. Similarly, included are details of how to generate all embeddings of Class 3 segments about each common head vertex, including accounting for

descendants of those segments which may also terminate at that common vertex; however, no method has yet been identified to implement the creation and manipulation of the data structures involved to perform this operation in linear time and memory and this remains as an open question from this work.

Chapter 5 details the algorithm relating it back to the earlier chapters and to the similarities and differences with Hopcroft and Tarjan's work; there are several differences:

- The main difference is that Hopcroft and Tarjan's algorithm ensures that each path is embedded onto the left (inside) stack which results in additional flips within the data structure that may be unnecessary; this then may result in segments being moved that can only be embedded within the inside region (relative to its parent) to the right stack breaking the affordance between the stacks and the regions and making it much harder to generate an embedding. The algorithm presented here maintains the affordance between the stacks and the regions by relaxing the embedding process and allowing segments to be embedded in both the inside and outside regions (as appropriate); this keeps the natural affordance between stacks and regions and simplifies the embedding process.
- Another change is that the data structures are maintained as the embedding is generated. By having a list for *partially embedded* and a list of all embedded segments within a block then, as per H&T, the *partially embedded* segments can be appended and discarded from the data structure while their presence is maintained in the list of all segments thus preserving the embedding information for use when generating a cyclic edge order. This change can be made independently of the other variations between algorithms and could equally be applied to H&T's original algorithm.
- The final difference (and main emphasis of this work) is in the identification of multiple embeddings. By identifying that the transformations of the data structure in H&T's algorithm are performing Whitney flips within the embedding and identifying cases where these are restricted from occurring then this has lead to an algorithm that can identify all possible embeddings of a biconnected graph (and all components of connected graphs so future work can be performed on an efficient means on combining these components).

Chapter 5 also investigates the upper time bound for the algorithm concluding that for a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ with segments $\mathcal{S}$ then:

- Trémaux Tree and segment generation have an $O(\mathcal{V} + \mathcal{E})$ upper time bound; and
- Planarity testing and permuting the data structure for the embedding have an $O(\mathcal{S})$ upper time bound (where $O(\mathcal{S}) \leq O(\mathcal{E})$).
- The generation of a cyclic edge order has $O(\mathcal{S} + \mathcal{E})$ upper time bound.

For all connected graphs, both the number of vertices and the number of segments is less than one plus the number of edges. Therefore, these upper time bounds can be simplified to all being $O(\mathcal{E})$ and the conclusion drawn is that the theoretical time bound for execution of this algorithm is linear.

Chapter 7 contains empirical testing of the algorithm to test this theorised time bound. The empirical testing shows that in all cases a linear best-fit provides the best approximation to the results when the graph has more than 750 vertices. This linear best-fit has, in both the empirical results for this algorithm and in Hopcroft and Tarjan's paper [32], a negative value for the y-intercept – no conclusion is drawn to the reason for this occurrence but a negative

execution time is clearly impossible and so it is not surprising that the empirical tests for small graph sizes do not fit to a linear best-fit.

The overall conclusion from these results indicate that the theoretical linear time bound for the algorithm is not invalidated by the empirical testing; so the algorithm is accepted to have a linear time bound.

8.1 Future Work

Several areas of interest have been identified during this work that can benefit from further work:

The one area where a linear time bounded implementation has not been identified is the generation of all possible embeddings of multiple Class 3 segments about a common head vertex. A sketch of a technique to achieve this result has been included on page 66 but an efficient data structure has not been identified to manage the reorganisation of the segments, especially when those Class 3 segments have descendants that also terminate at that common head vertex.

The algorithm presented is intended as a demonstration of the principles of performing this test and embedding; it has not been optimised and various strategies can be considered:

- It should be possible to generate the segments during backtracking of the initial DFS search and then bucket sort the segments to order each segment's children rather than bucket sorting the edges and subsequently generating the segments; this should improve the efficiency of the first two phases of the algorithm.
- One avenue of investigation is to consider whether a cyclic edge order can be generated during the embedding of the segments. This could be considered by trying to integrate a PQ-Tree data structure into the embedding such that *Q-nodes* are used to represent the embedding of *flippable* segments/blocks and *P-nodes* are used to represent groups of *permutable* segments. A thorough investigation needs to be performed on the relationship between the embedding of segments and the transformations of the cyclic edge orders to determine if this data structure is actually appropriate as it is not immediately clear if this is the case.

In considering the utility of this algorithm, these could include: those that are interested in testing for the existence of specific faces within an embedding (chemical compositions); testing an embedding against metrics to find an optimal embedding (circuit board design or graphical design of layouts); or to find a specific embedding (graph isomorphism). In all these cases, the segment and block hierarchy is constant once a graph is embedded and by divorcing the generation of an embedding and the associated blocks from the generation of a cyclic edge order allows for the interesting possibility of considering the parallelisation of the process of generating and testing large numbers of embeddings against specific metrics. One can also consider whether it is possible to improve on a brute force search when looking to test embeddings of a graph against specific metrics by considering whether performing specific flips and permutations can achieve a gradual improvement against a metric or if restricting certain flips or permutations can retain desired features in an embedding.

The final area for future work is that the empirical testing has show that variations in the root vertex and in the graph structure give differing characteristics to the best-fits. Of particular interest is the results shown in figure A.3.2 (page 209) which shows a step in the graph and there is a corresponding change in the properties of the graph shown in figure 7.6.4 (page 162).

A more detailed analysis of different graph structures could be performed to identify structures which require more or less time to test and embed and what structures cause the step change in execution time.

Appendix A

Empirical Testing Figures

This appendix chapter contains plots of various result generated during empirical testing. Each plot presents best-fits for curves of different shapes against the timing data from the empirical testing for specific input conditions (number of vertices and edges of the graph and choice of root vertex and graph structure); whilst the best-fit curves are displayed graphically, the curves are often overlapping (or close to it) making distinguishing the features of the curve difficult and so the pertinent information is displayed numerically in the legend of each plot.

The results can be partitioned into three groups:

- Variation in the choice of root vertex whilst keeping the graph size and structure constant;
- Variation in the graph size (number of vertices, and proportionally, the number of edges and segments) whilst keeping the graph structure and, proportional to the size, the choice of root vertex constant; and
- Variation in the number of edges of the graph whilst keeping the number of vertices, graph structure and root vertex constant.

The following plots fit into one of the above categories and are included in the appendix because it is difficult to elicit a quantitative comparison of the best-fit curve in the same plot using the graphical representation and so the important information on the best-fit curves is the numerical data contained in the legend which can be summarised in the main body of this work without overburdening it by adding multiple pages of plots. The plots do, however, show some additional features and anomalies and, where pertinent, a single example is drawn into the main body of work to highlight these whilst additional examples remain here.

The matlab code used to generate the plots contained here, and in the main body of work, is included in Appendix E.

A.1 Variations in Root Vertex for Constant Graph Size and Type

A.1.1 Triangular Prism Graphs

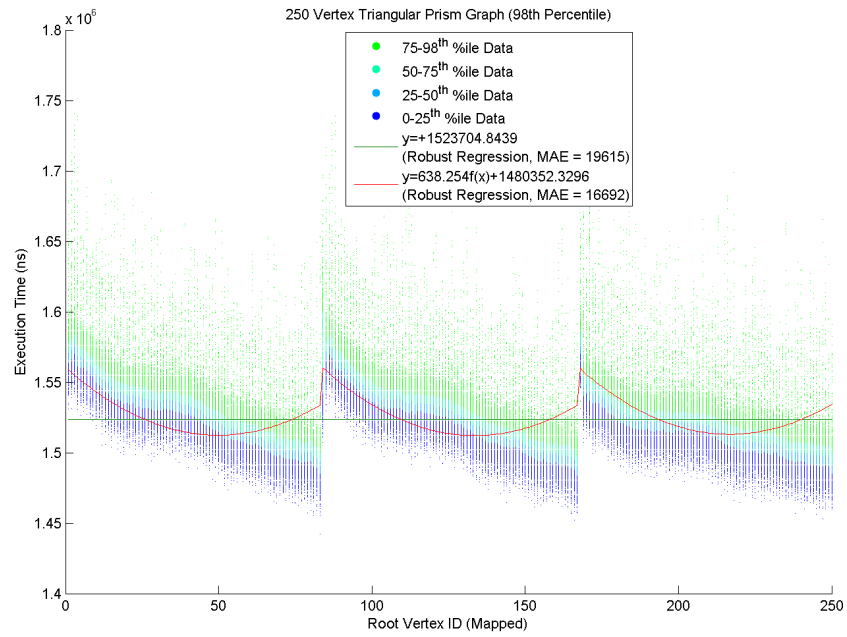


Figure A.1.1: Execution Times for Varying Root Vertices of 250 Vertex Triangular Prism Graph

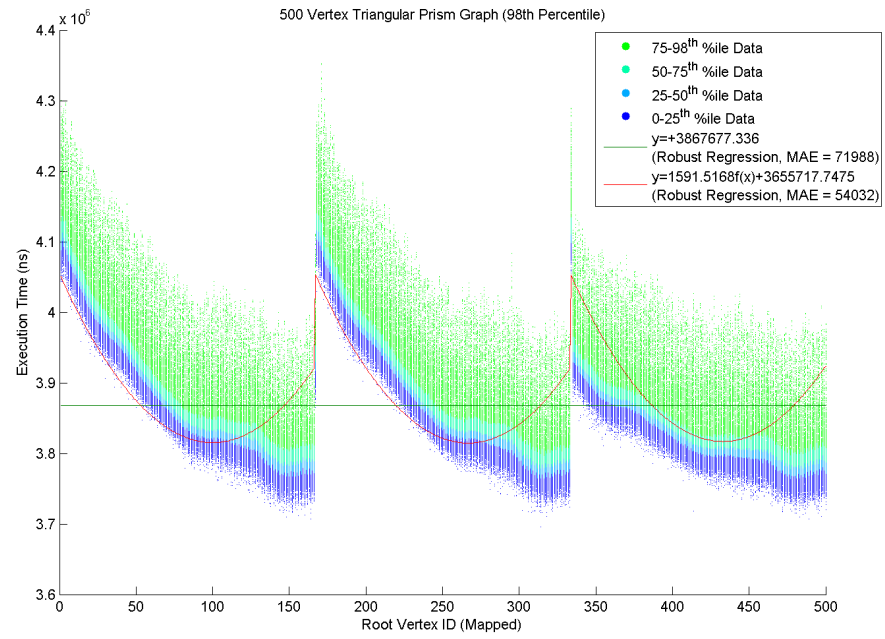


Figure A.1.2: Execution Times for Varying Root Vertices of 500 Vertex Triangular Prism Graph

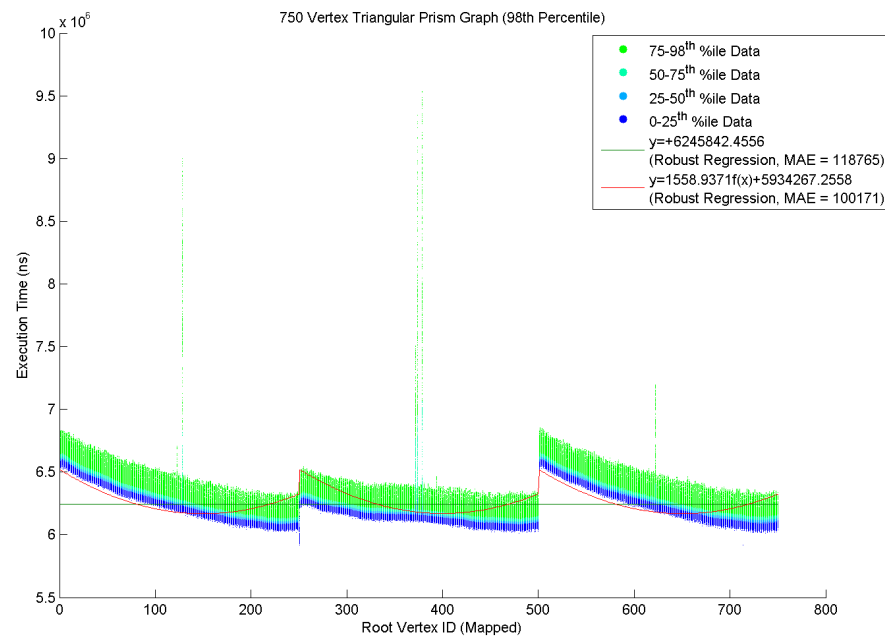


Figure A.1.3: Execution Times for Varying Root Vertices of 750 Vertex Triangular Prism Graph

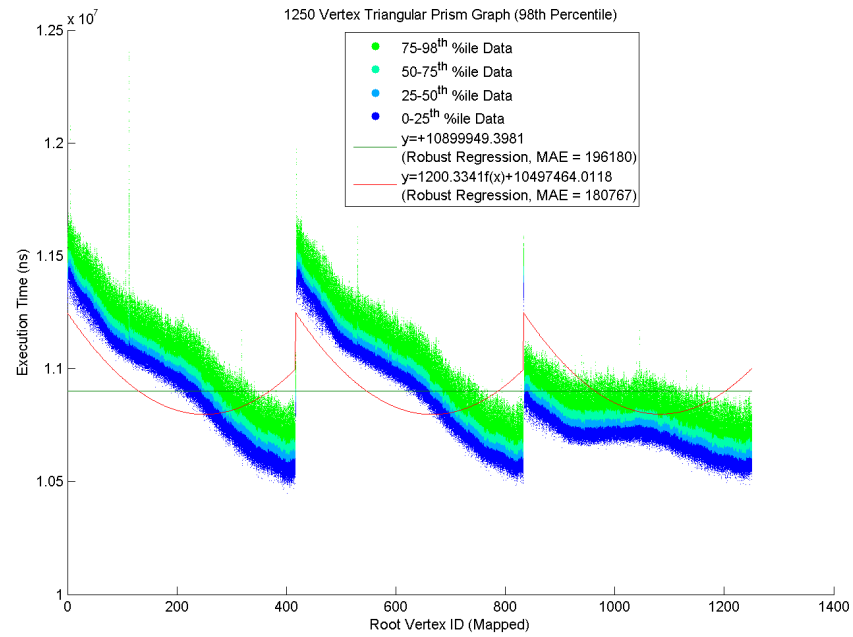


Figure A.1.4: Execution Times for Varying Root Vertices of 1250 Vertex Triangular Prism Graph

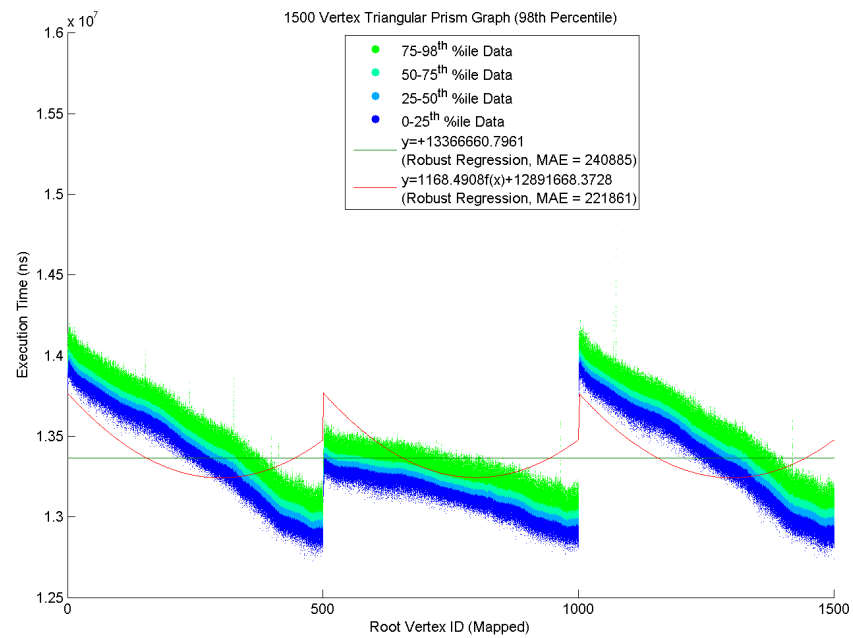


Figure A.1.5: Execution Times for Varying Root Vertices of 1500 Vertex Triangular Prism Graph

A.1.2 Planar Wheel Graphs

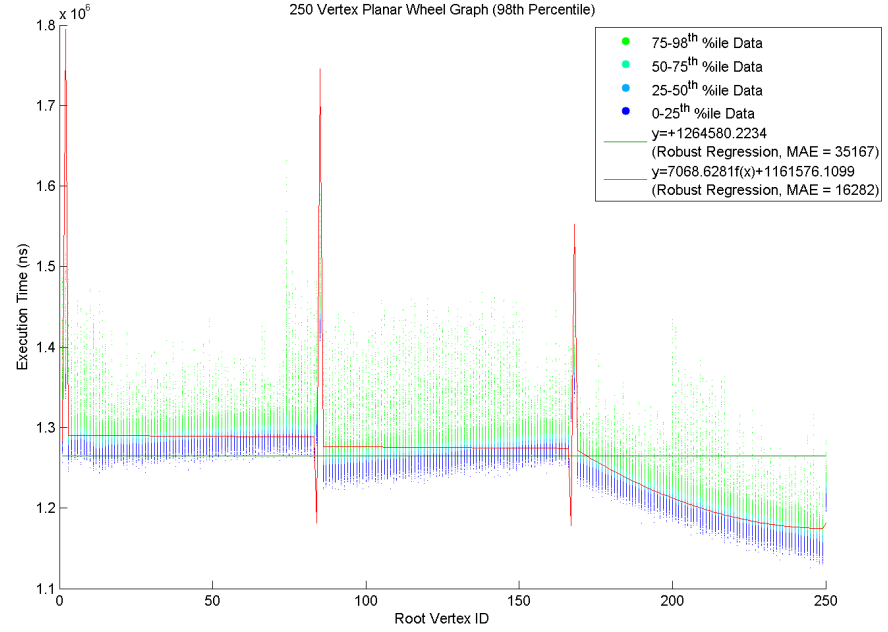


Figure A.1.6: Execution Times for Varying Root Vertices of 250 Vertex Planar Wheel Graph

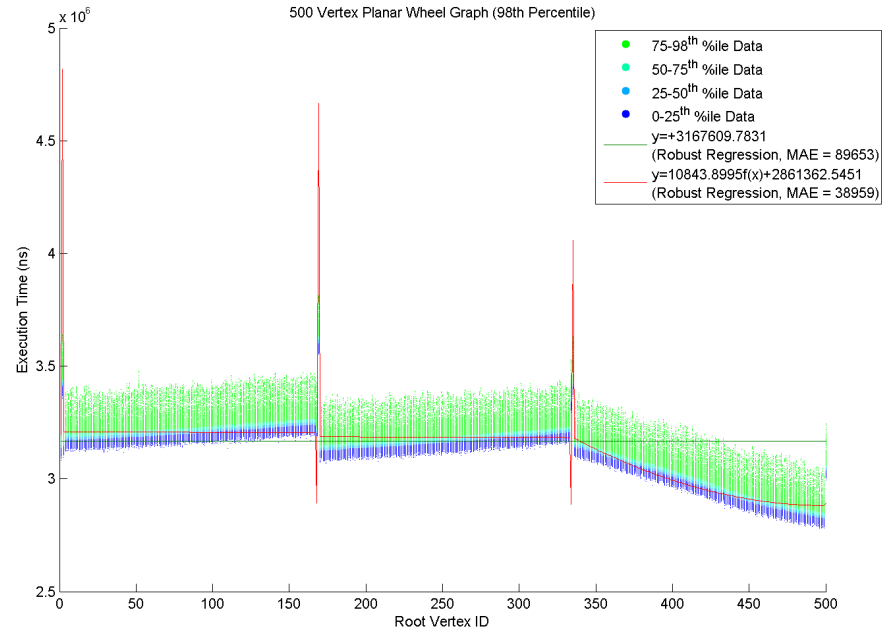


Figure A.1.7: Execution Times for Varying Root Vertices of 500 Vertex Planar Wheel Graph

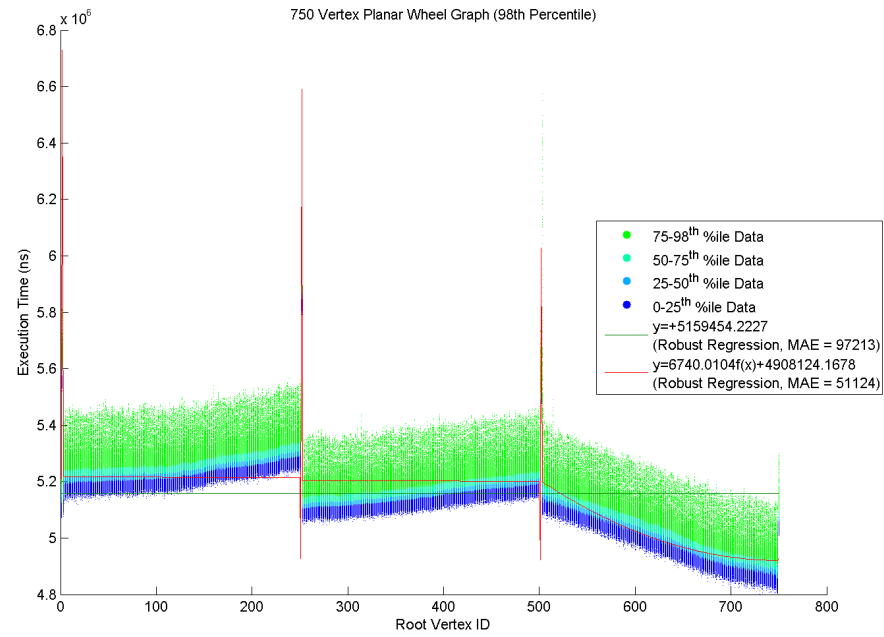


Figure A.1.8: Execution Times for Varying Root Vertices of 750 Vertex Planar Wheel Graph

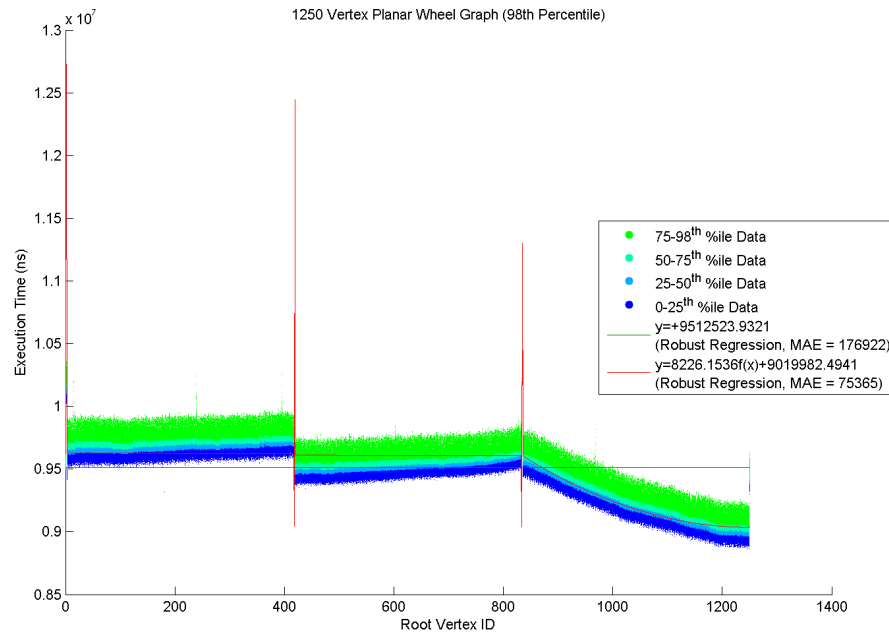


Figure A.1.9: Execution Times for Varying Root Vertices of 1250 Vertex Planar Wheel Graph

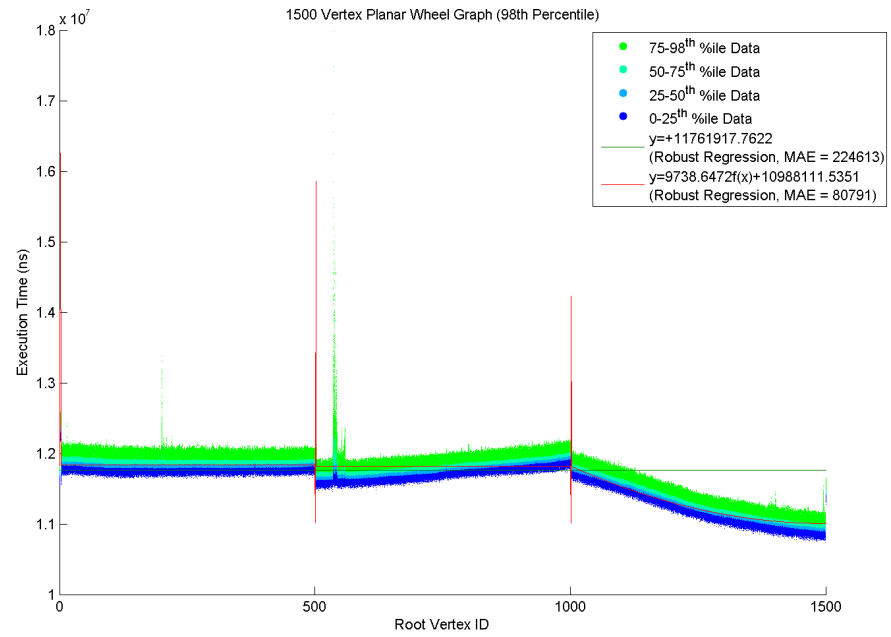


Figure A.1.10: Execution Times for Varying Root Vertices of 1500 Vertex Planar Wheel Graph

A.1.3 Ordered Face Trisection Graphs

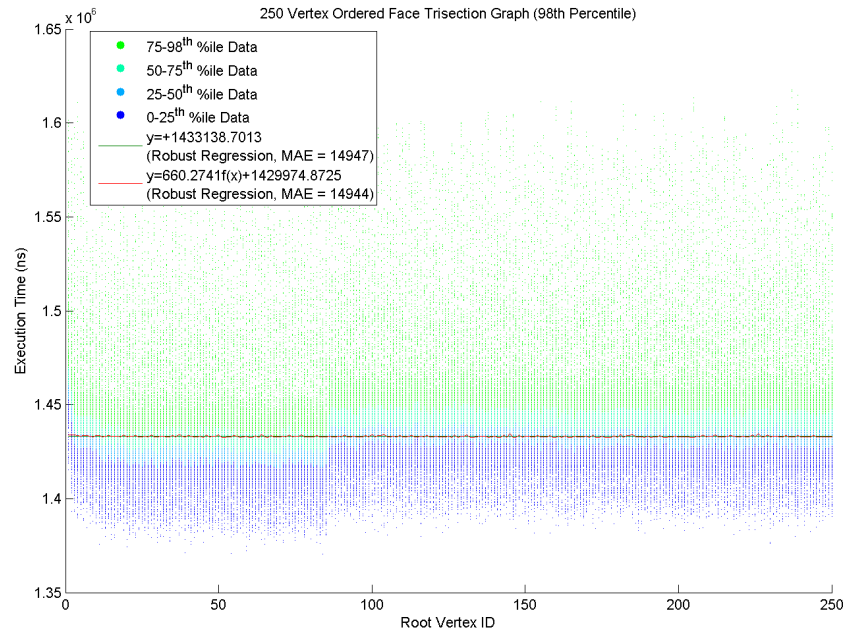


Figure A.1.11: Execution Times for Varying Root Vertices of 250 Vertex Ordered Face Trisection Graph

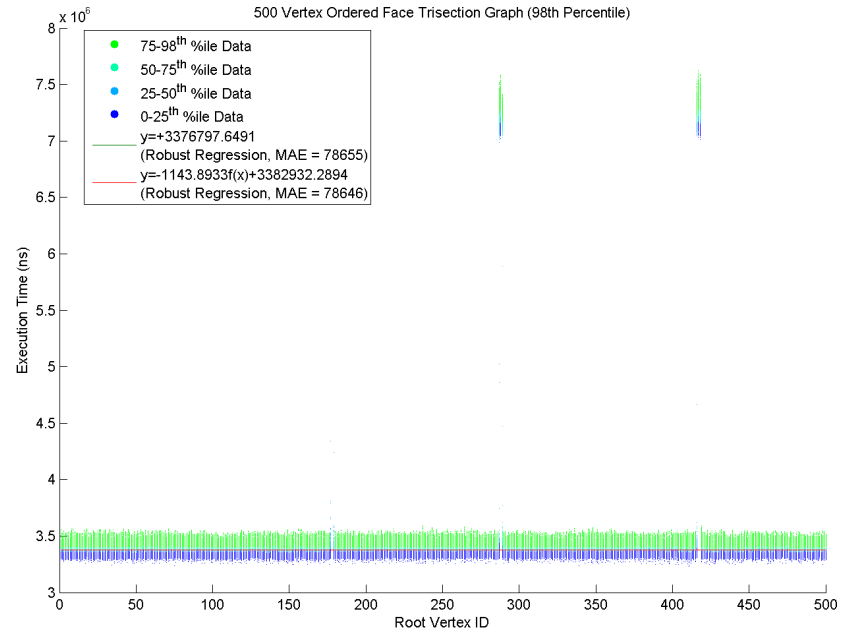


Figure A.1.12: Execution Times for Varying Root Vertices of 500 Vertex Ordered Face Trisection Graph

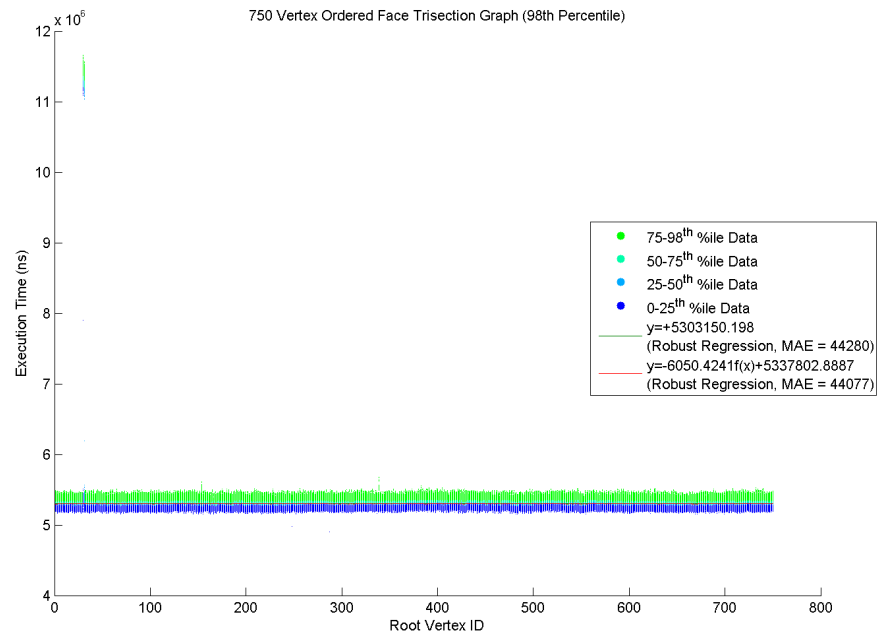


Figure A.1.13: Execution Times for Varying Root Vertices of 750 Vertex Ordered Face Trisection Graph

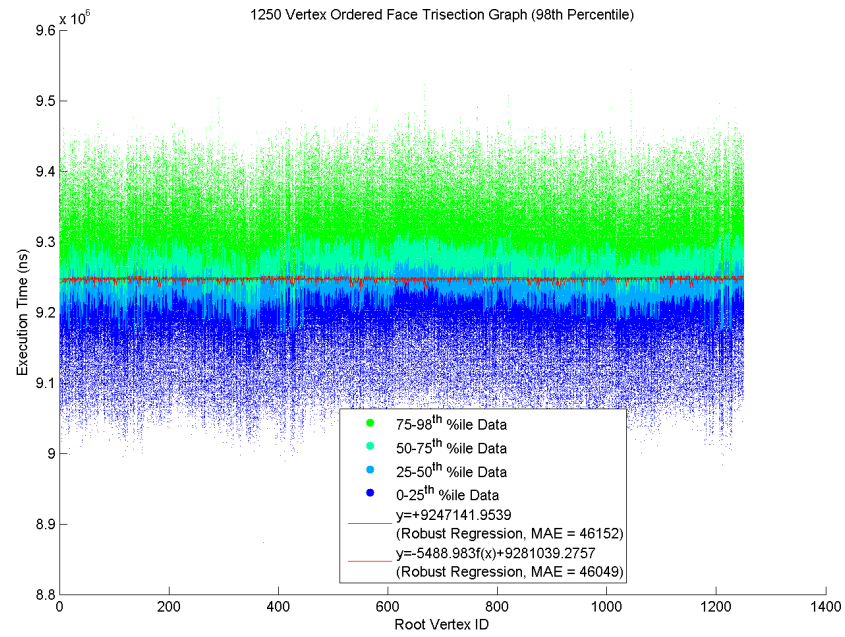


Figure A.1.14: Execution Times for Varying Root Vertices of 1250 Vertex Ordered Face Trisection Graph

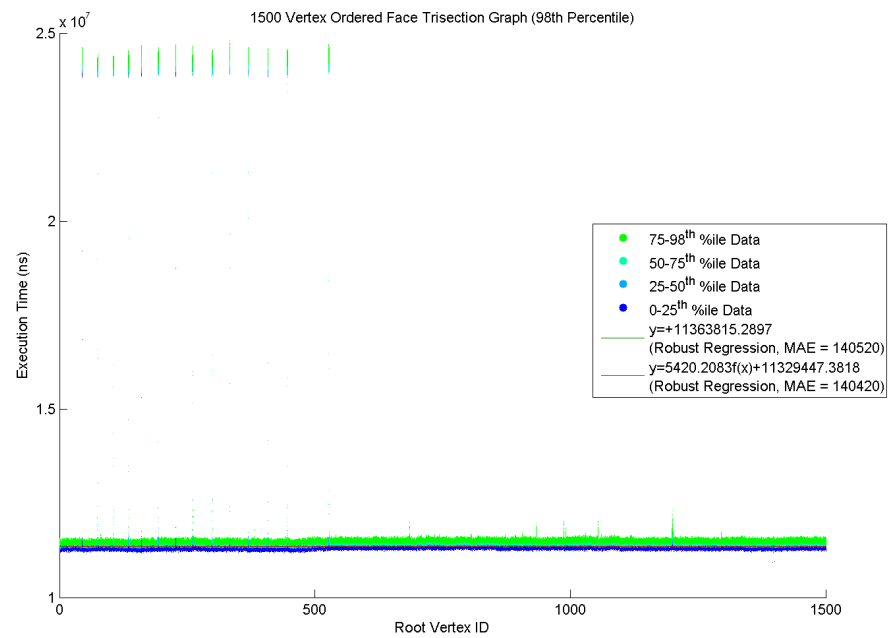


Figure A.1.15: Execution Times for Varying Root Vertices of 1500 Vertex Ordered Face Trisection Graph

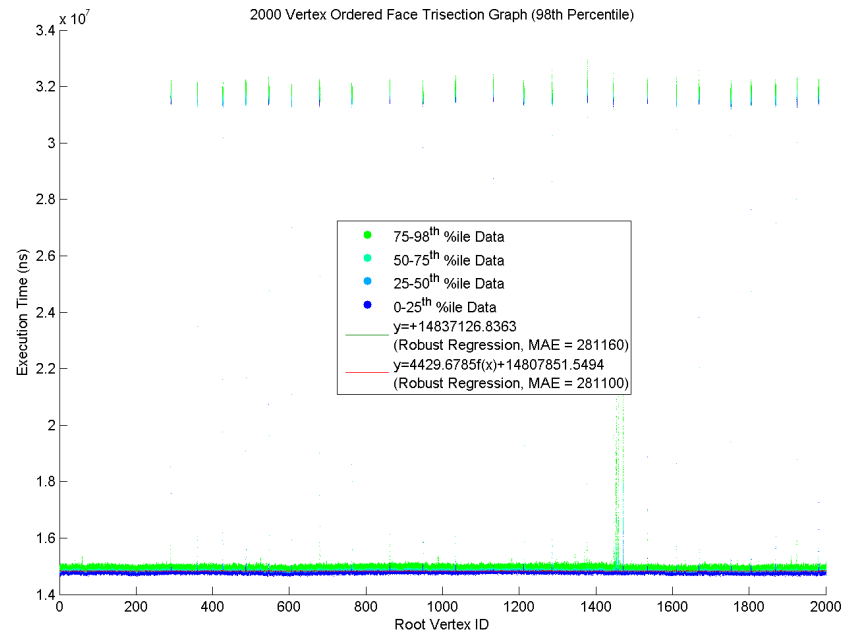


Figure A.1.16: Execution Times for Varying Root Vertices of 2000 Vertex Ordered Face Trisection Graph

A.1.4 Random Face Trisection Graphs

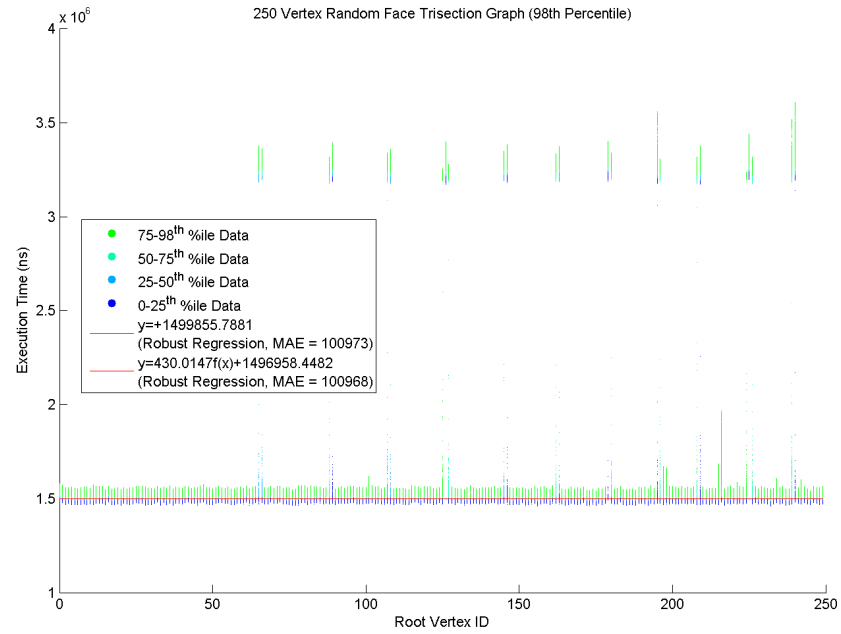


Figure A.1.17: Execution Times for Varying Root Vertices of 250 Vertex Random Face Trisection Graph

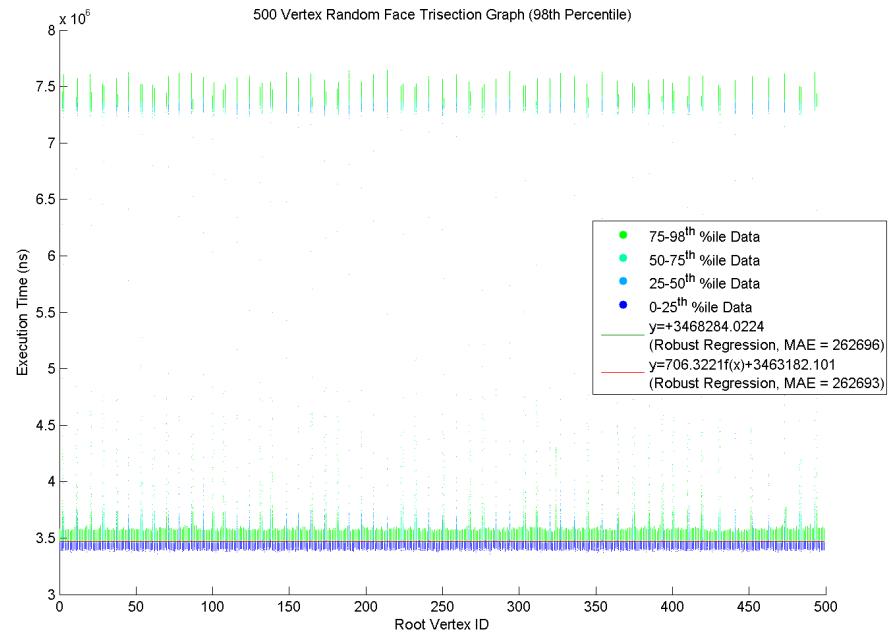


Figure A.1.18: Execution Times for Varying Root Vertices of 500 Vertex Random Face Trisection Graph

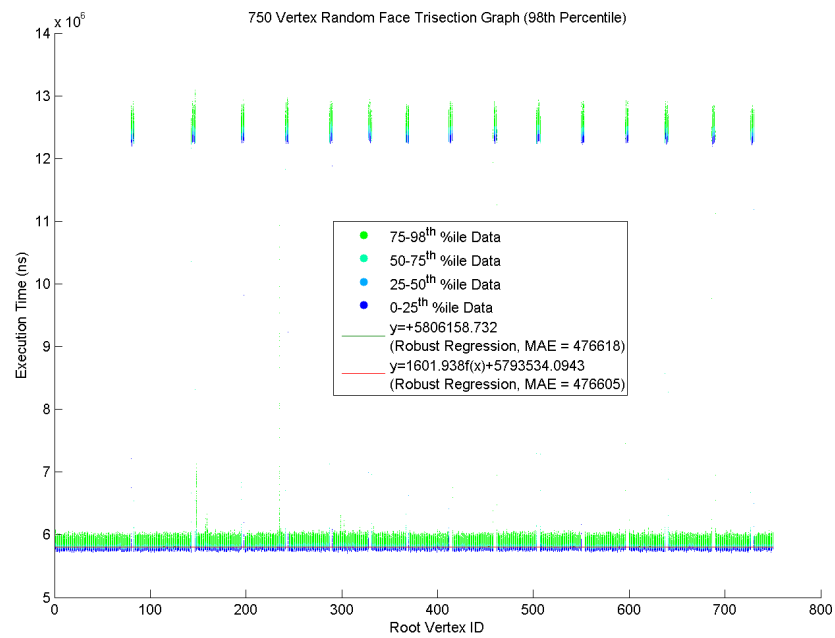


Figure A.1.19: Execution Times for Varying Root Vertices of 750 Vertex Random Face Trisection Graph

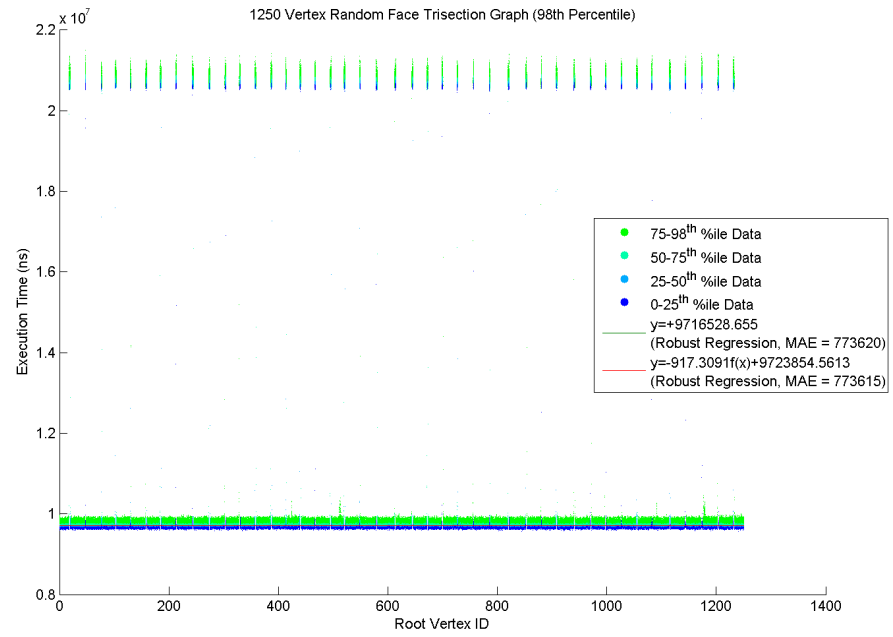


Figure A.1.20: Execution Times for Varying Root Vertices of 1250 Vertex Random Face Trisection Graph

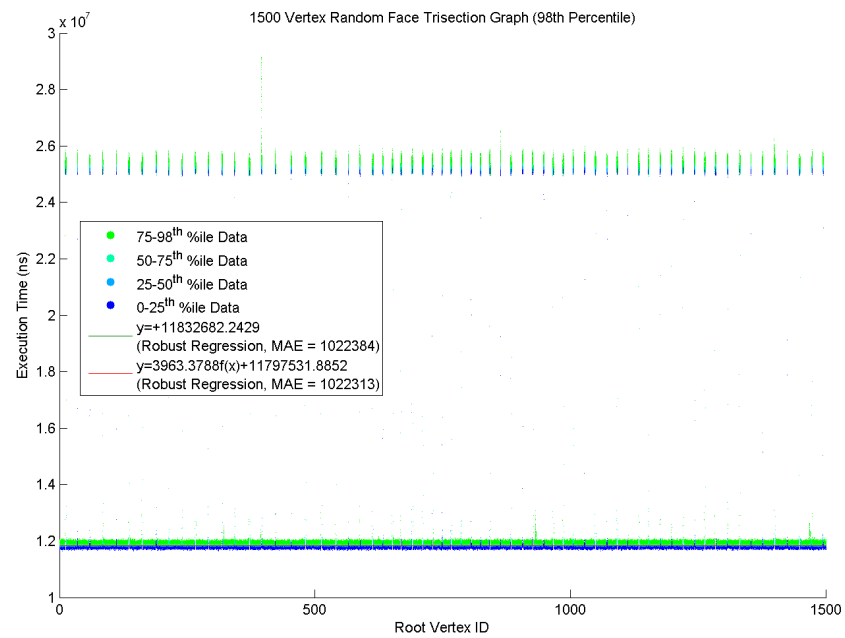


Figure A.1.21: Execution Times for Varying Root Vertices of 1500 Vertex Random Face Trisection Graph

A.2 Variation in Graph Size for Constant Graph Type and Root Vertex

A.2.1 Triangular Prism Graphs (42-4500 Vertices)

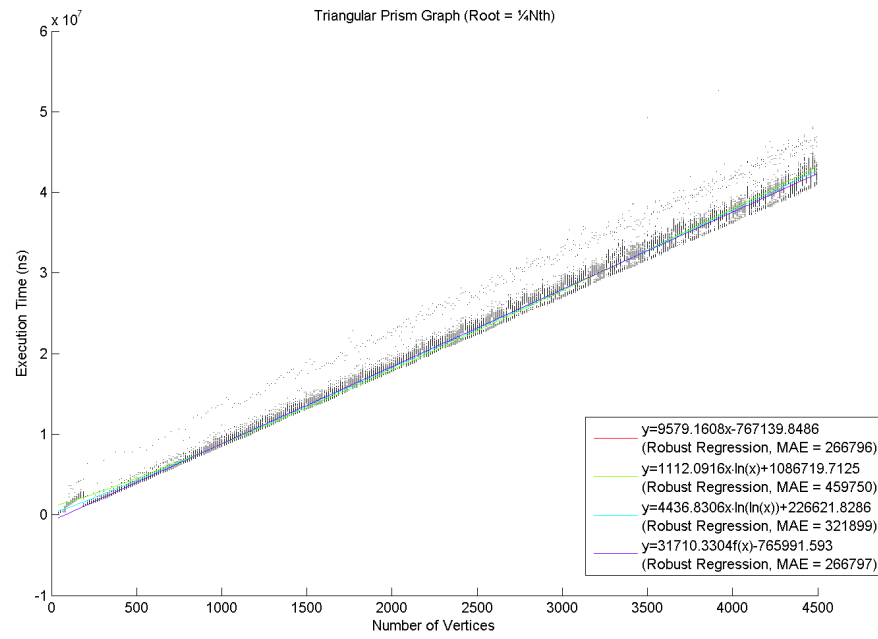
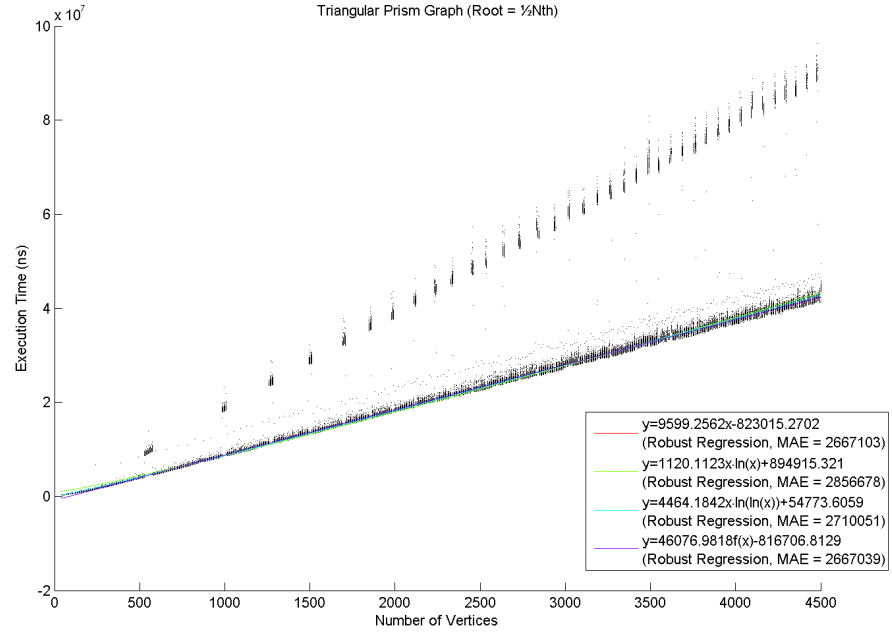
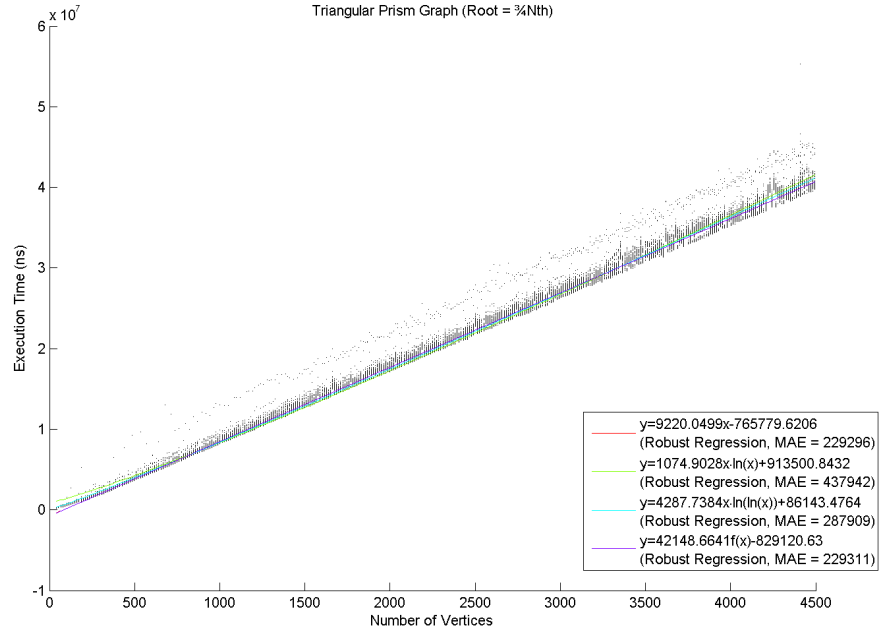
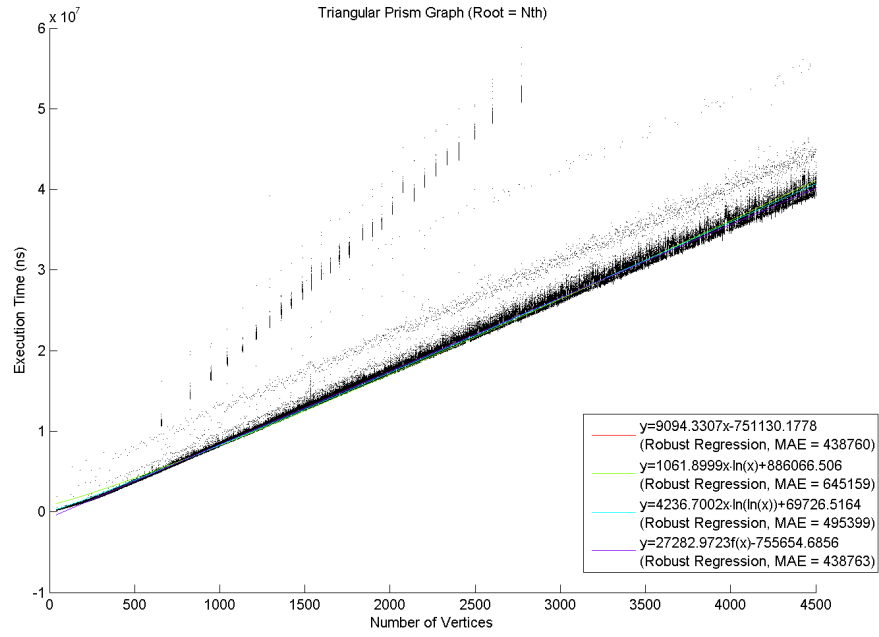
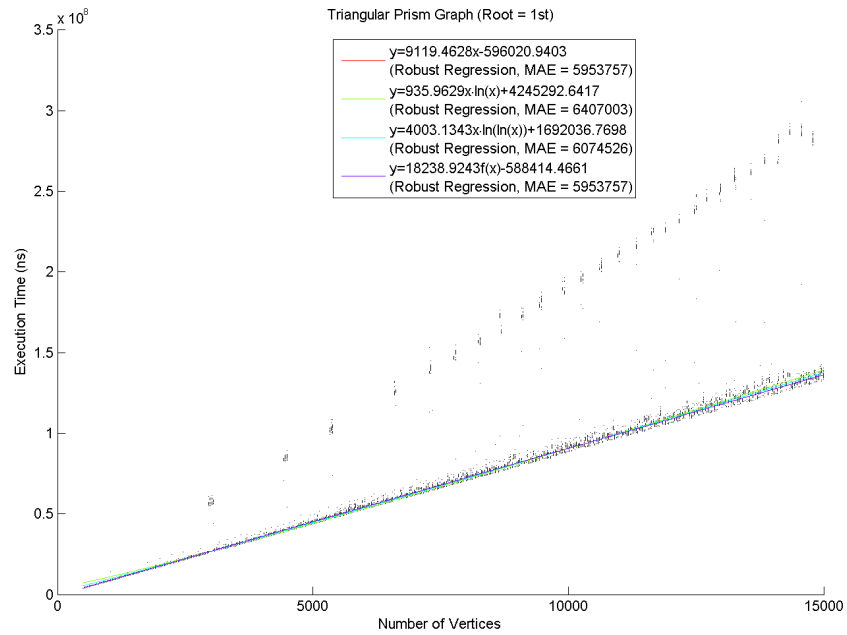


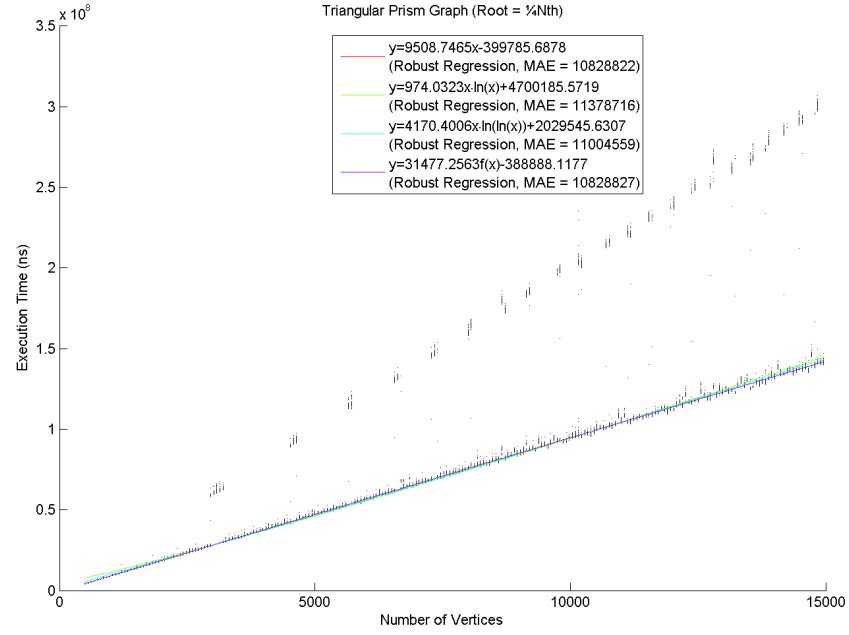
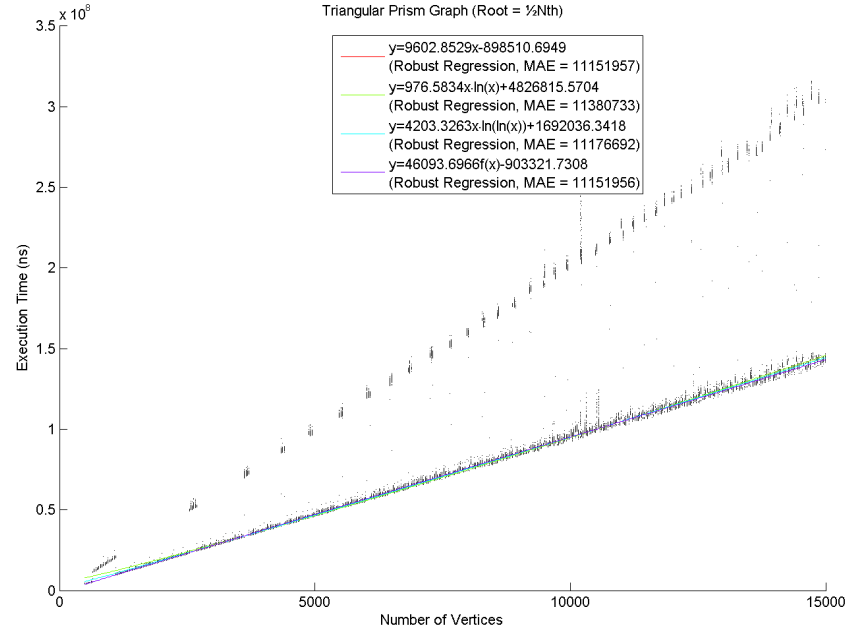
Figure A.2.1: Triangular Prism Graph (42-4500 Vertices, $\frac{1}{4}N^{\text{th}}$ Vertex Root)

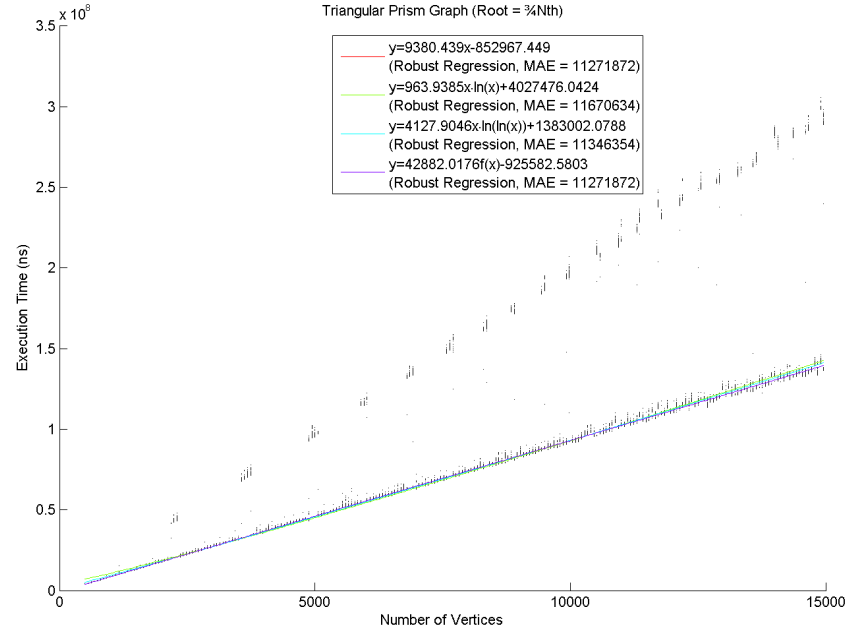
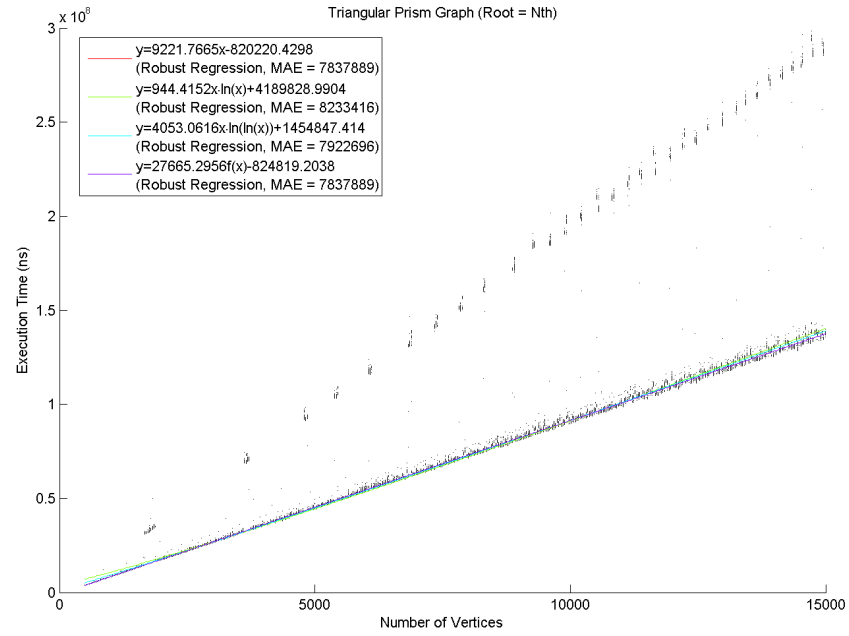
Figure A.2.2: Triangular Prism Graph (42-4500 Vertices, $\frac{1}{2}N^{\text{th}}$ Vertex Root)Figure A.2.3: Triangular Prism Graph (42-4500 Vertices, $\frac{3}{4}N^{\text{th}}$ Vertex Root)

Figure A.2.4: Triangular Prism Graph (42-4500 Vertices, N^{th} Vertex Root)

A.2.2 Triangular Prism Graphs (500-15000 Vertices)

Figure A.2.5: Triangular Prism Graph (500-15000 Vertices, 1st Vertex Root)

Figure A.2.6: Triangular Prism Graph (500-15000 Vertices, $\frac{1}{4}N^{\text{th}}$ Vertex Root)Figure A.2.7: Triangular Prism Graph (500-15000 Vertices, $\frac{1}{2}N^{\text{th}}$ Vertex Root)

Figure A.2.8: Triangular Prism Graph (500-15000 Vertices, $\frac{3}{4}N^{\text{th}}$ Vertex Root)Figure A.2.9: Triangular Prism Graph (500-15000 Vertices, N^{th} Vertex Root)

A.2.3 Planar Wheel Graphs (42-4500 Vertices)

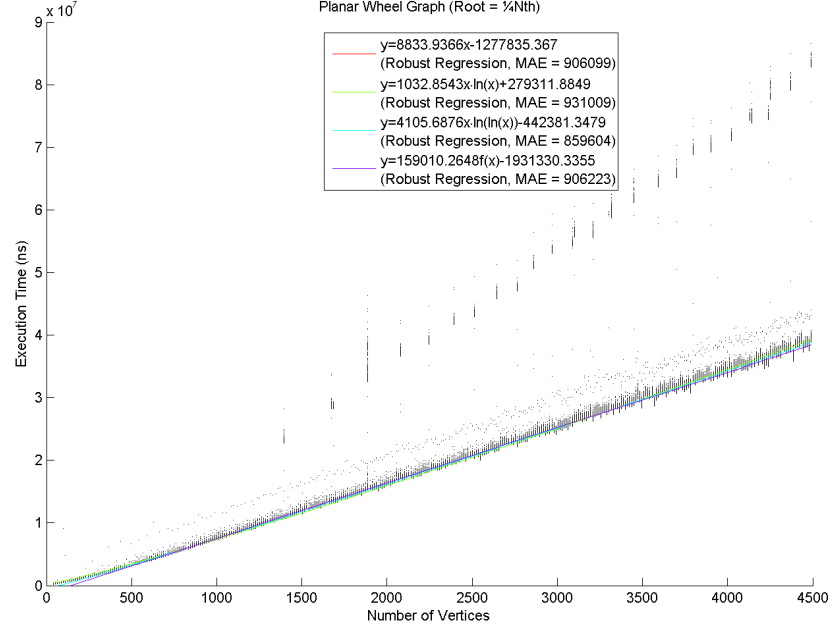


Figure A.2.10: Planar Wheel Graph (42-4500 Vertices, $\frac{1}{4}N^{\text{th}}$ Vertex Root)

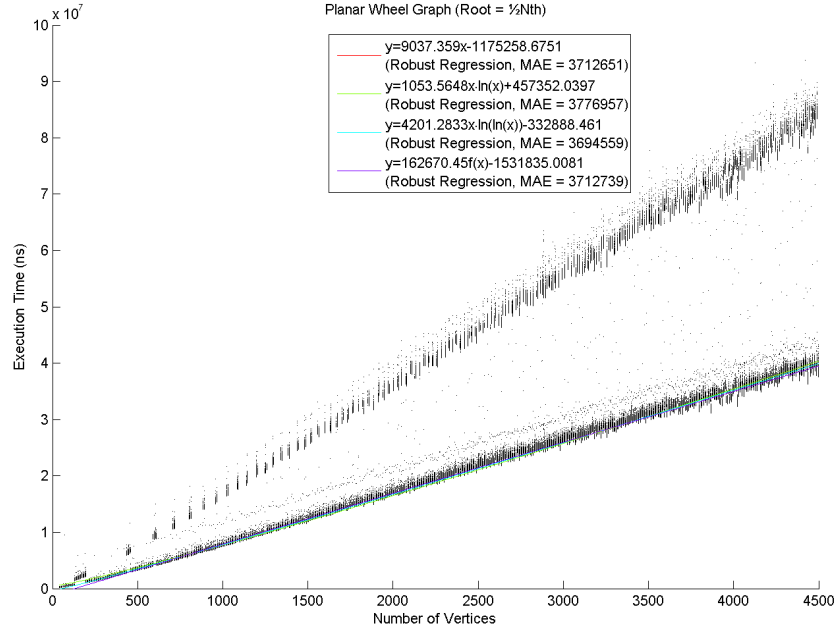
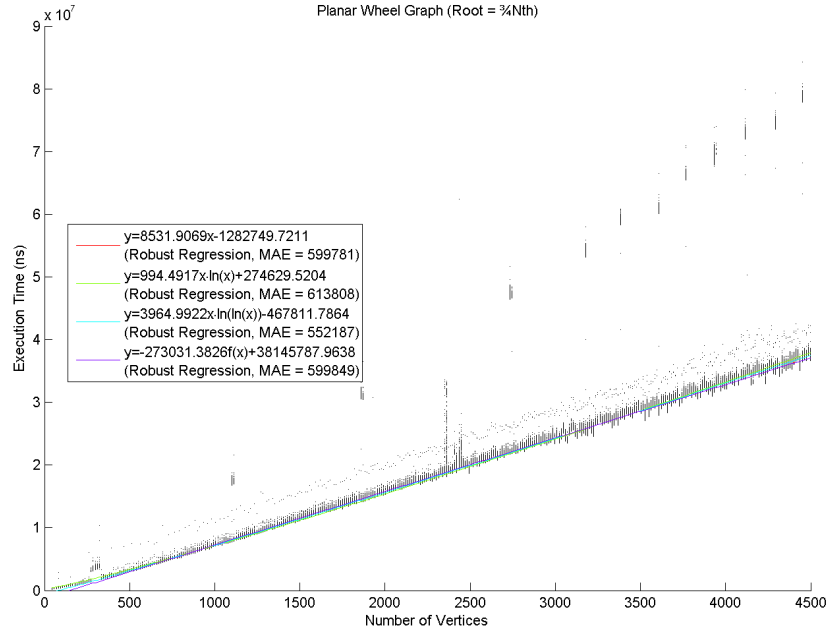
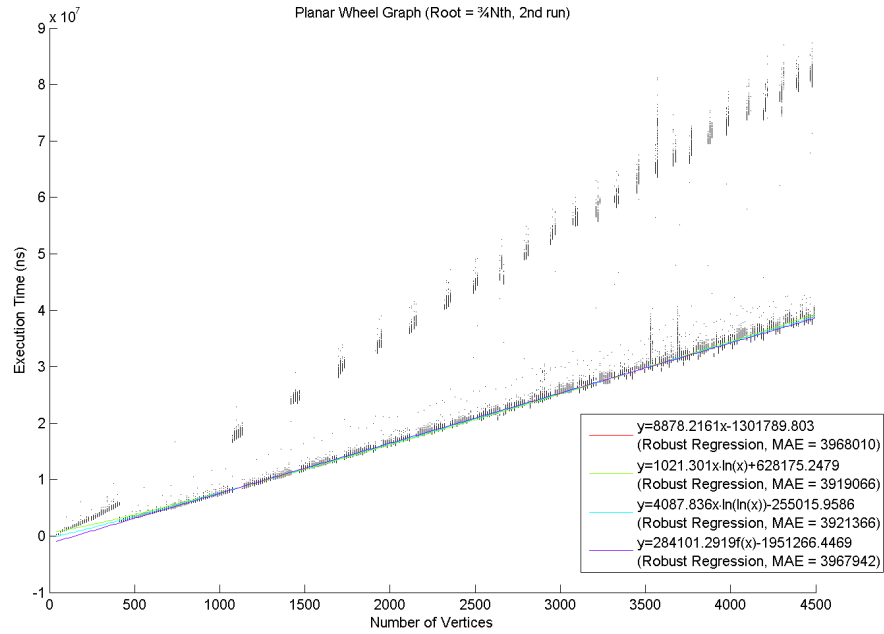
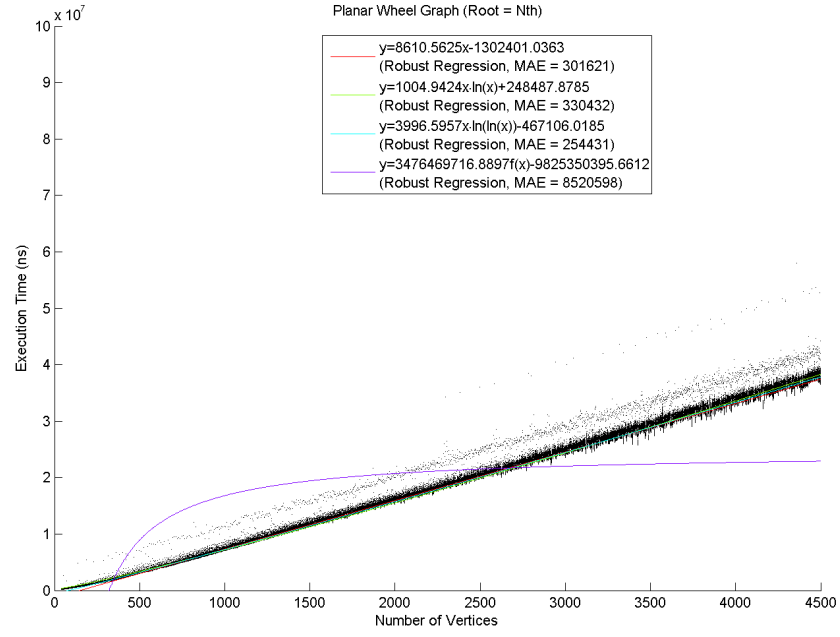
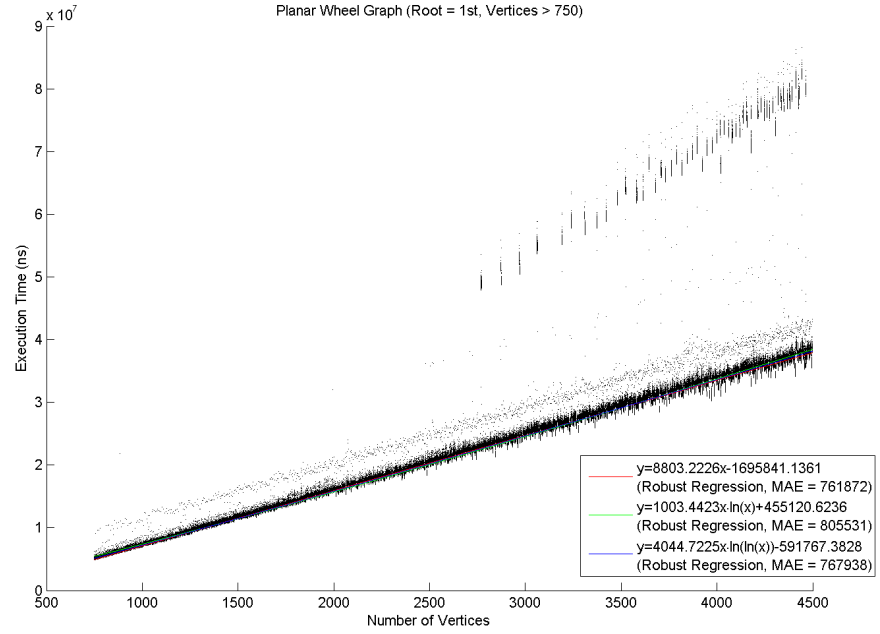
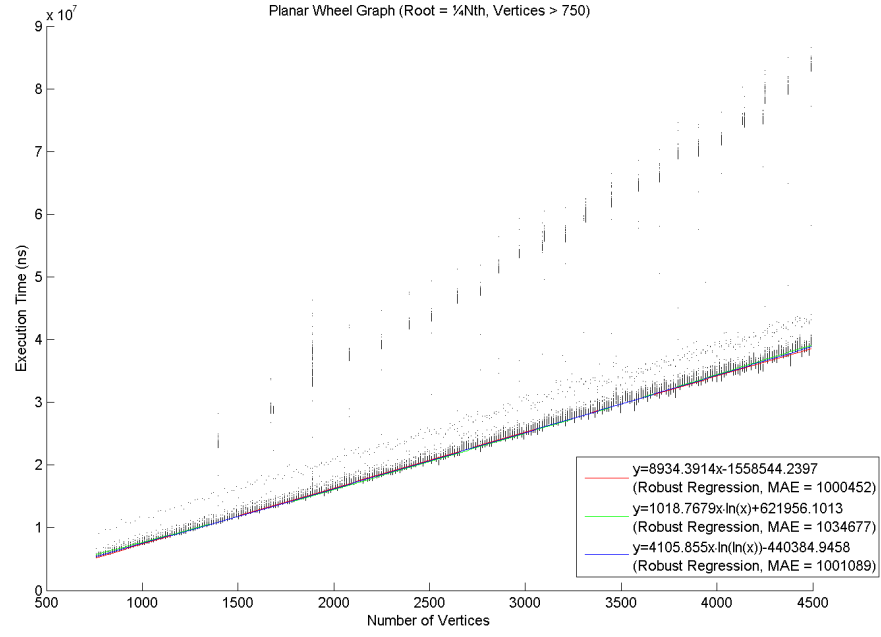
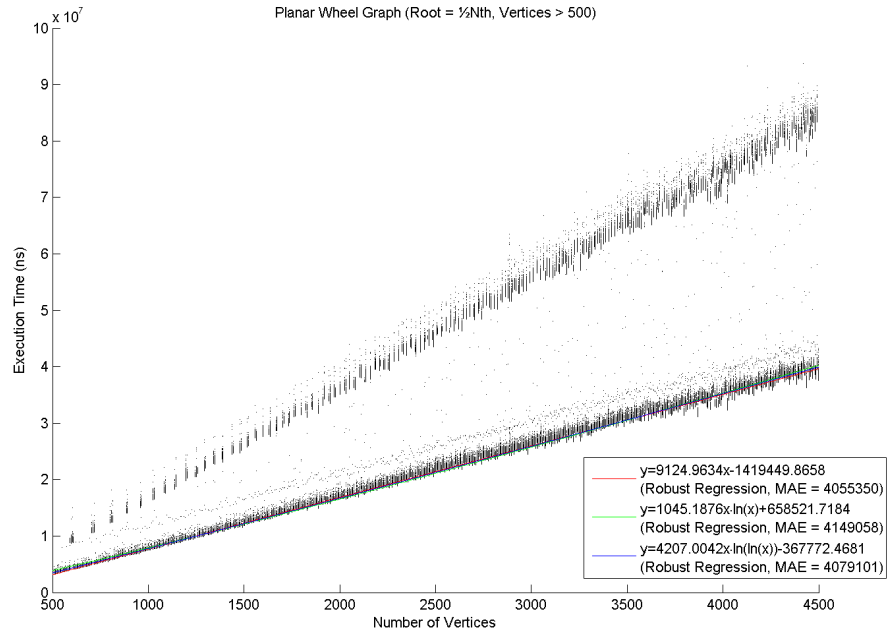
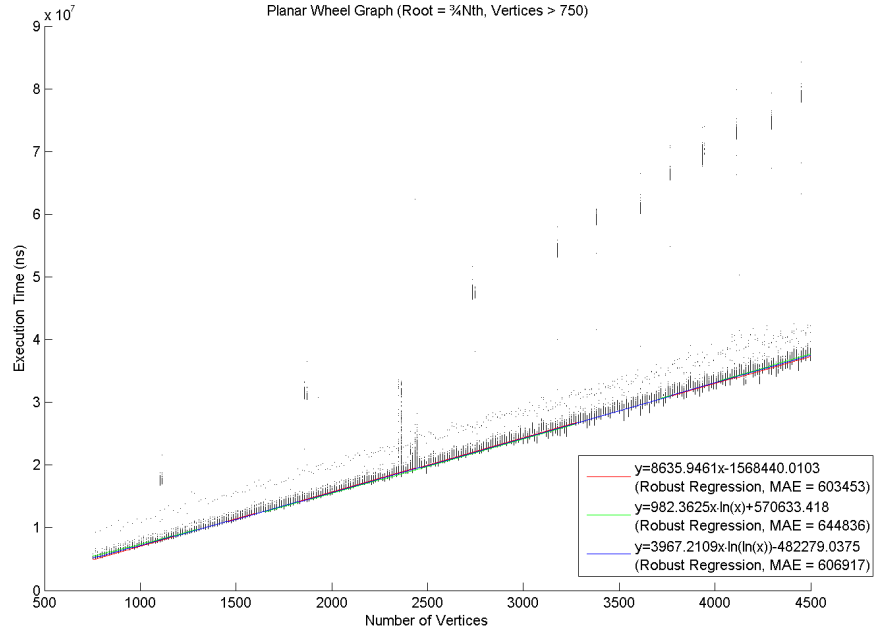
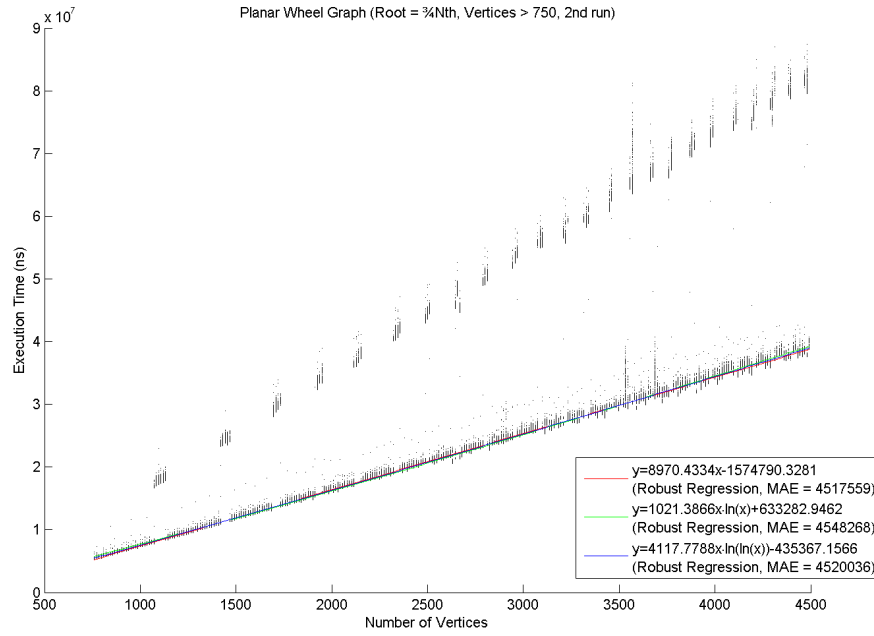


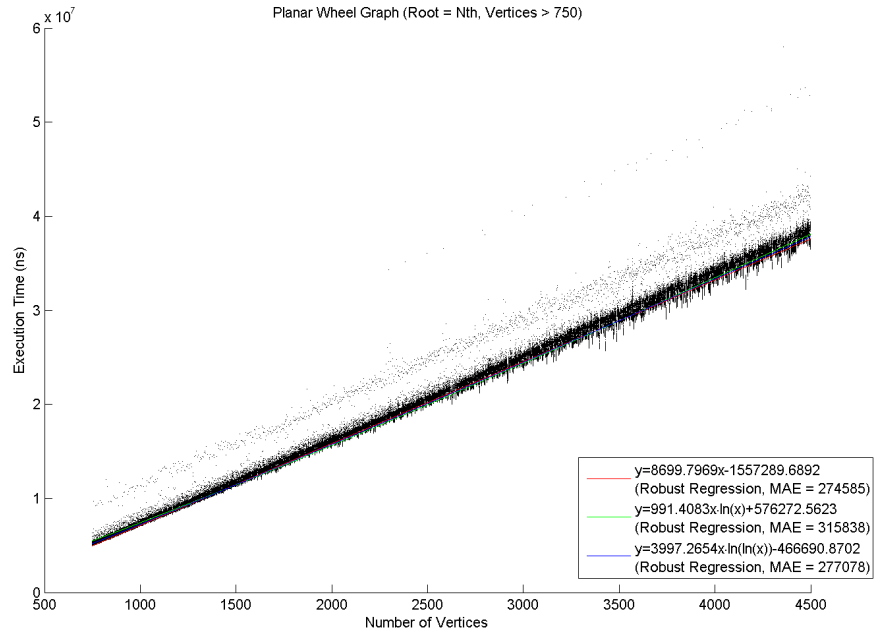
Figure A.2.11: Planar Wheel Graph (42-4500 Vertices, $\frac{1}{2}N^{\text{th}}$ Vertex Root)

Figure A.2.12: Planar Wheel Graph (42-4500 Vertices, $\frac{3}{4}N^{\text{th}}$ Vertex Root)Figure A.2.13: Planar Wheel Graph (42-4500 Vertices, $\frac{3}{4}N^{\text{th}}$ Vertex Root, 2nd Run)

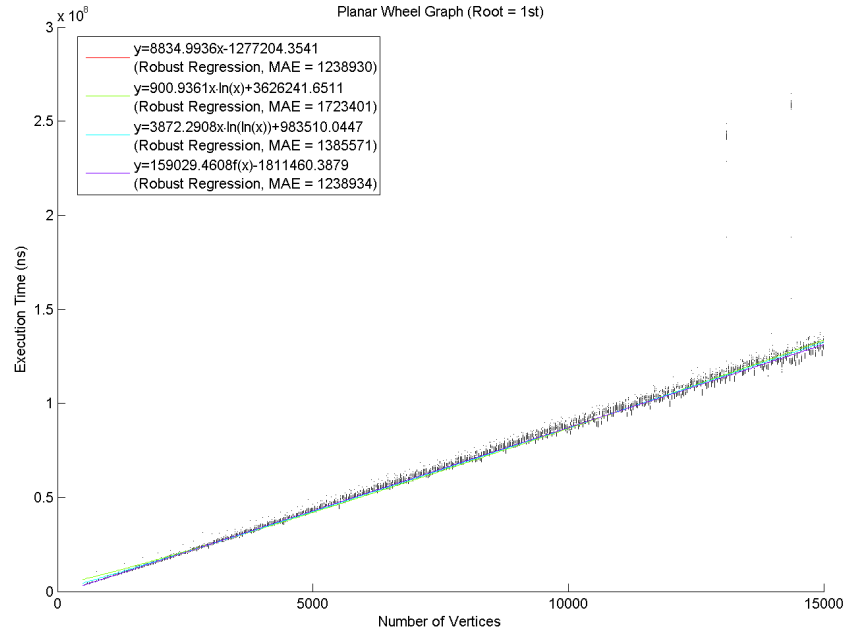
Figure A.2.14: Planar Wheel Graph (42-4500 Vertices, N^{th} Vertex Root)Figure A.2.15: Planar Wheel Graph (750-4500 Vertices, 1^{st} Vertex Root)

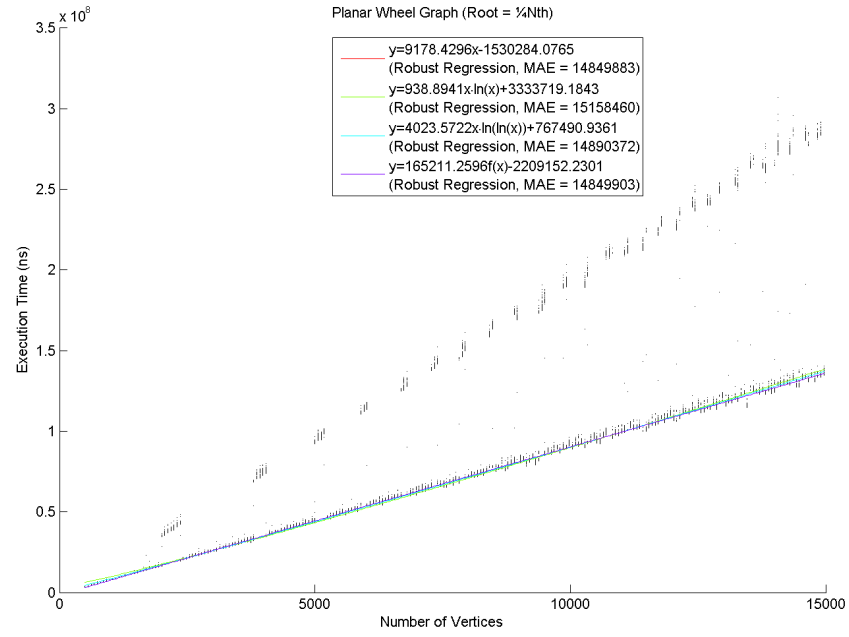
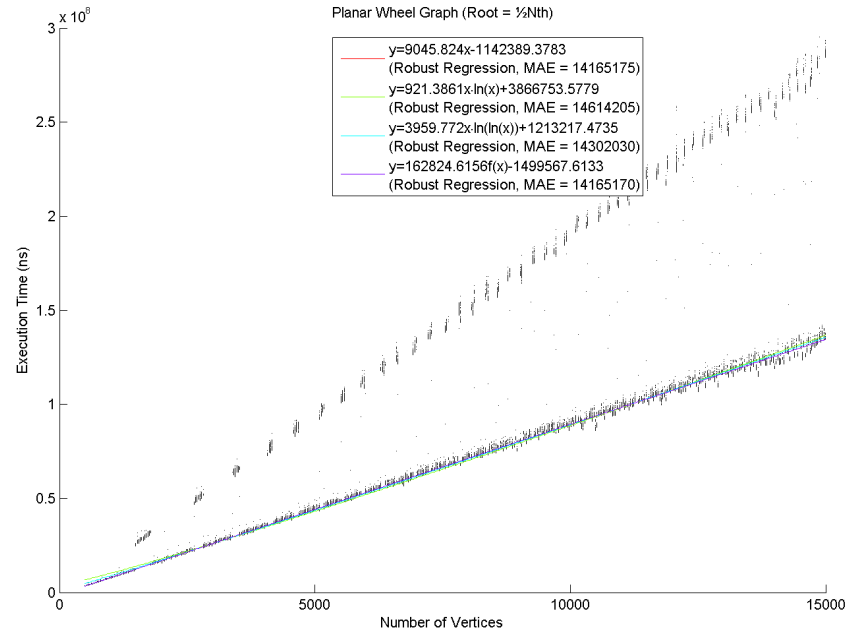
Figure A.2.16: Planar Wheel Graph (750-4500 Vertices, $\frac{1}{4}N^{\text{th}}$ Vertex Root)Figure A.2.17: Planar Wheel Graph (500-4500 Vertices, $\frac{1}{2}N^{\text{th}}$ Vertex Root)

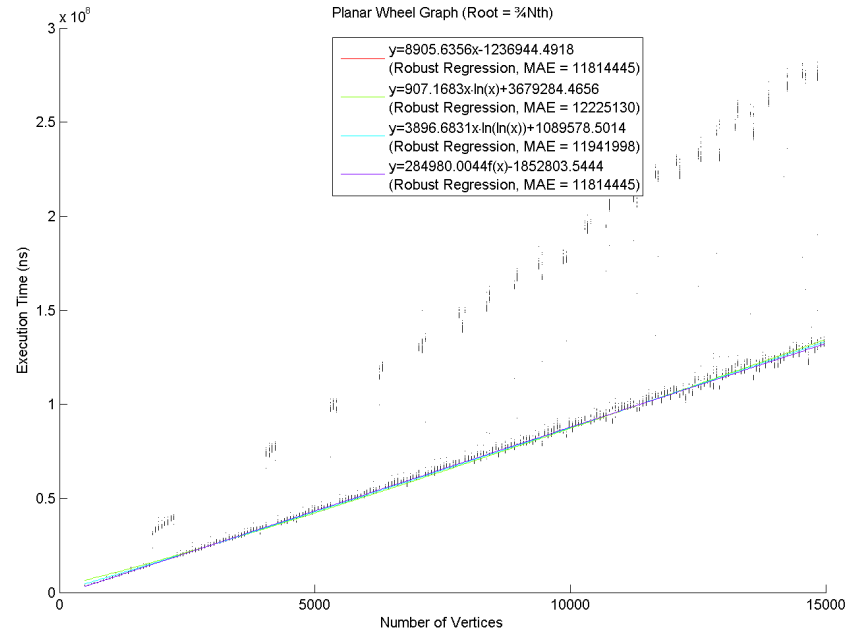
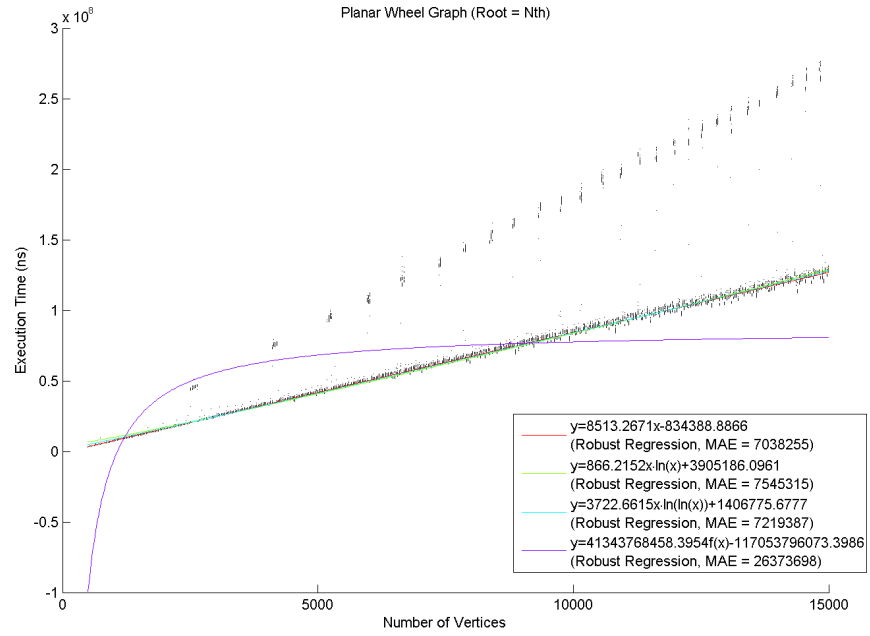
Figure A.2.18: Planar Wheel Graph (750-4500 Vertices, $\frac{3}{4}N^{\text{th}}$ Vertex Root)Figure A.2.19: Planar Wheel Graph (750-4500 Vertices, $\frac{3}{4}N^{\text{th}}$ Vertex Root, 2nd Run)

Figure A.2.20: Planar Wheel Graph (750-4500 Vertices, N^{th} Vertex Root)

A.2.4 Planar Wheel Graphs (500-15000 Vertices)

Figure A.2.21: Planar Wheel Graph (500-15000 Vertices, 1st Vertex Root)

Figure A.2.22: Planar Wheel Graph (500-15000 Vertices, $\frac{1}{4}N^{\text{th}}$ Vertex Root)Figure A.2.23: Planar Wheel Graph (500-15000 Vertices, $\frac{1}{2}N^{\text{th}}$ Vertex Root)

Figure A.2.24: Planar Wheel Graph (500-15000 Vertices, $\frac{3}{4}N^{\text{th}}$ Vertex Root)Figure A.2.25: Planar Wheel Graph (500-15000 Vertices, N^{th} Vertex Root)

A.2.5 Ordered Face Trisection Graphs (42-4500 Vertices)

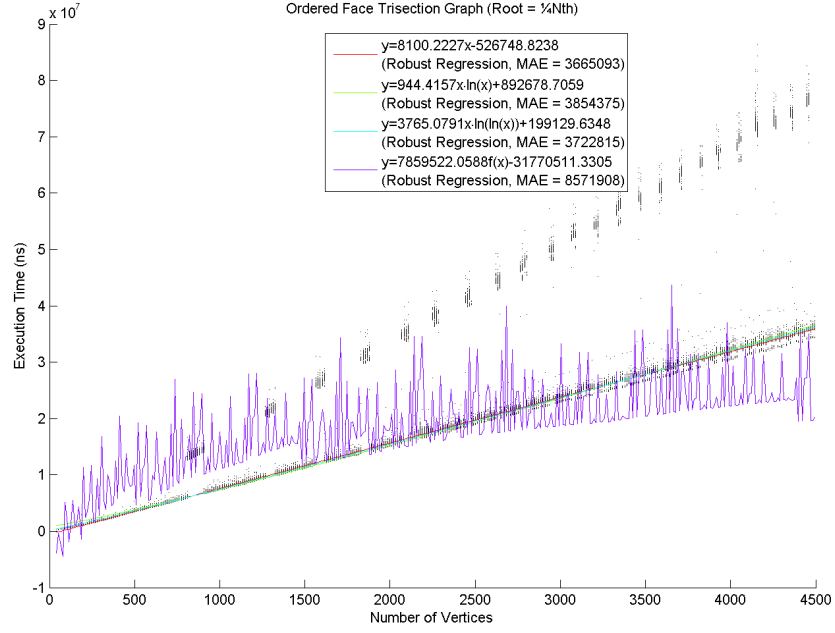


Figure A.2.26: Ordered Face Trisection Graph (42-4500 Vertices, $\frac{1}{4}N^{\text{th}}$ Vertex Root)

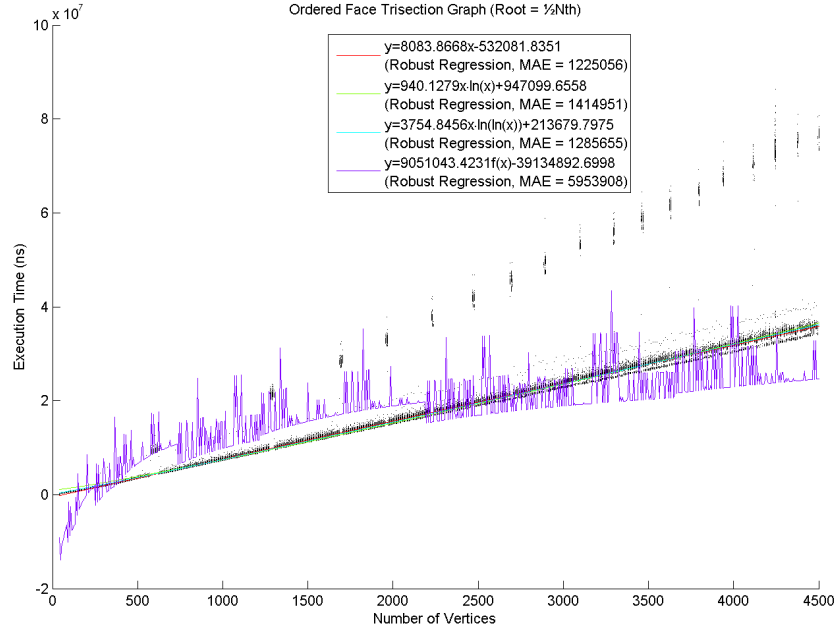
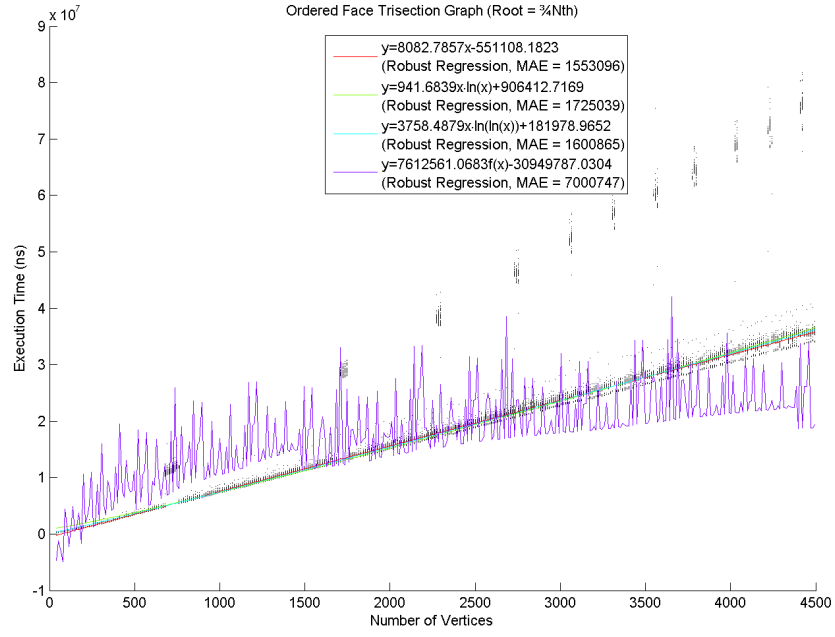
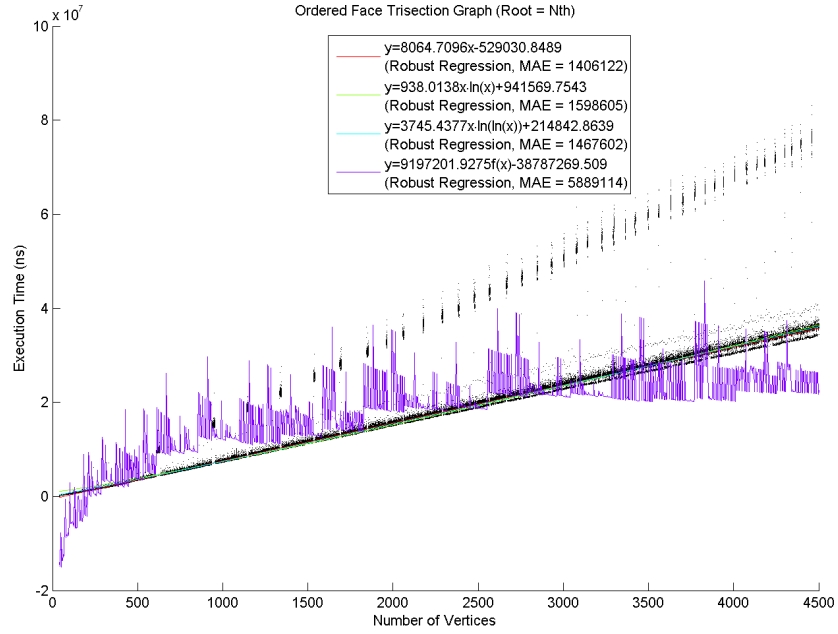


Figure A.2.27: Ordered Face Trisection Graph (42-4500 Vertices, $\frac{1}{2}N^{\text{th}}$ Vertex Root)

Figure A.2.28: Ordered Face Trisection Graph (42-4500 Vertices, $\frac{3}{4}N^{\text{th}}$ Vertex Root)Figure A.2.29: Ordered Face Trisection Graph (42-4500 Vertices, N^{th} Vertex Root)

A.2.6 Ordered Face Trisection Graphs (500-15000 Vertices)

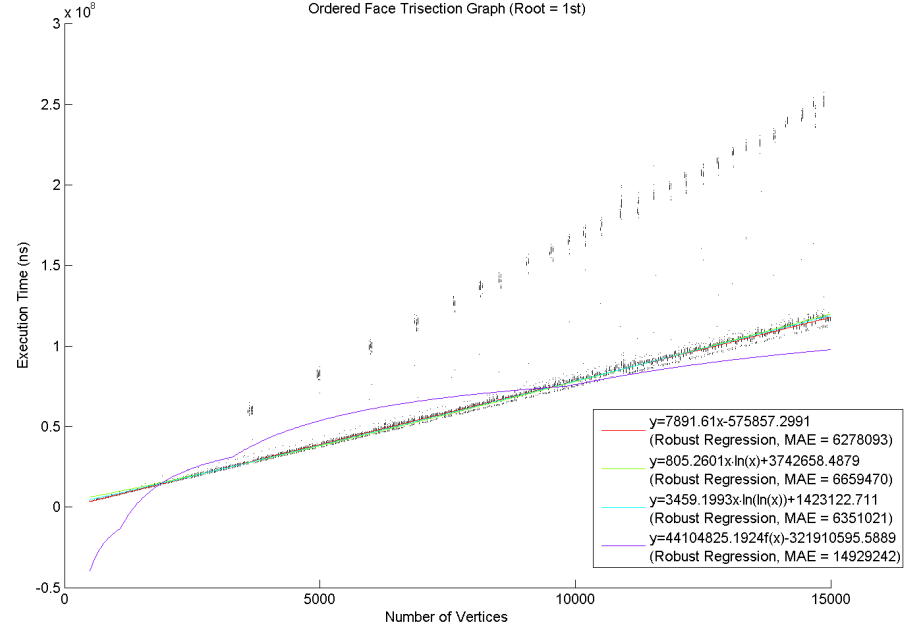


Figure A.2.30: Ordered Face Trisection Graph (500-15000 Vertices, 1st Vertex Root)

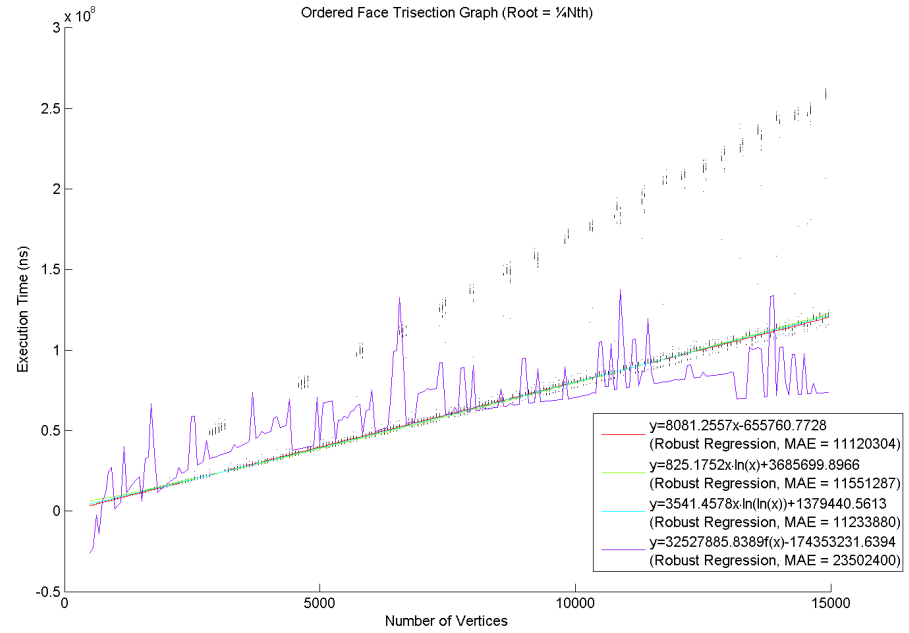
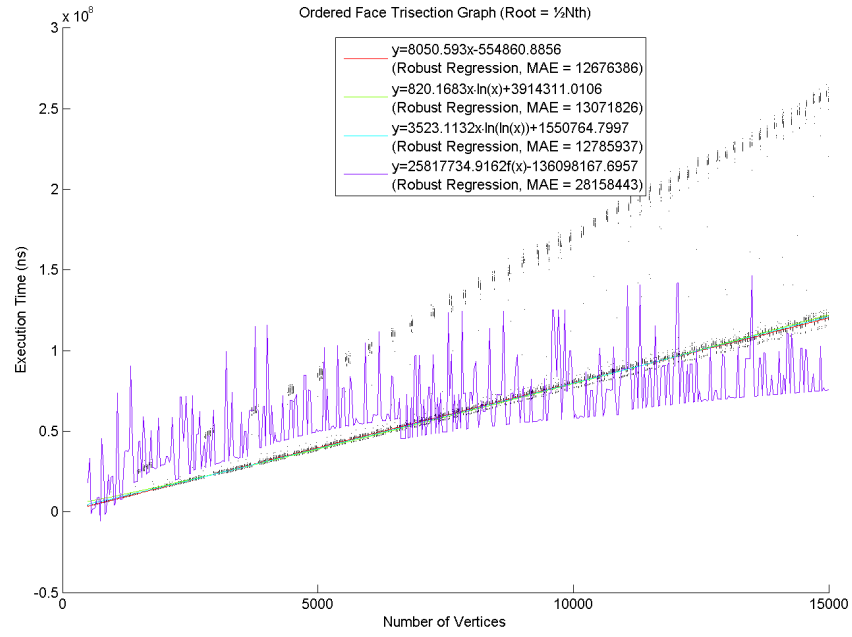
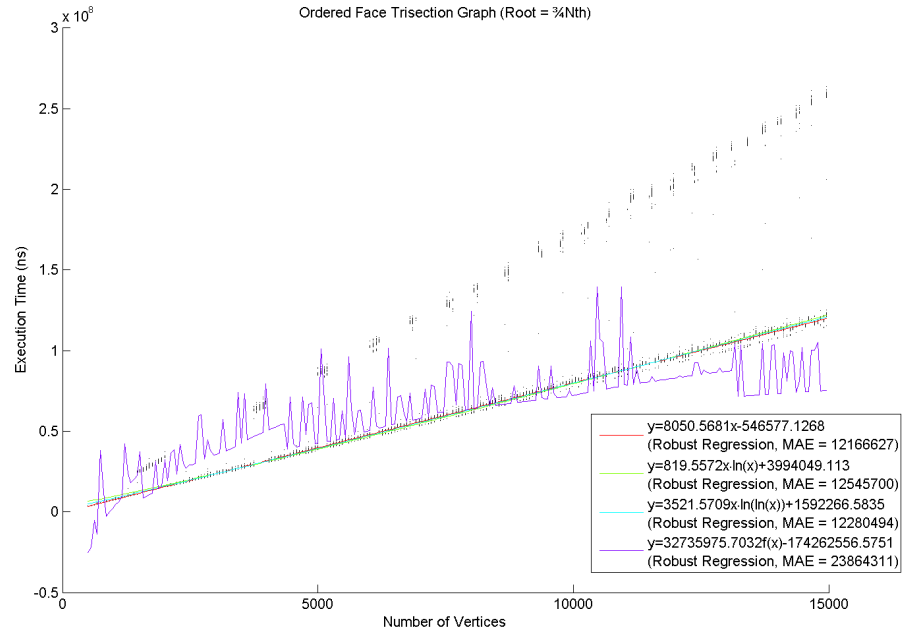
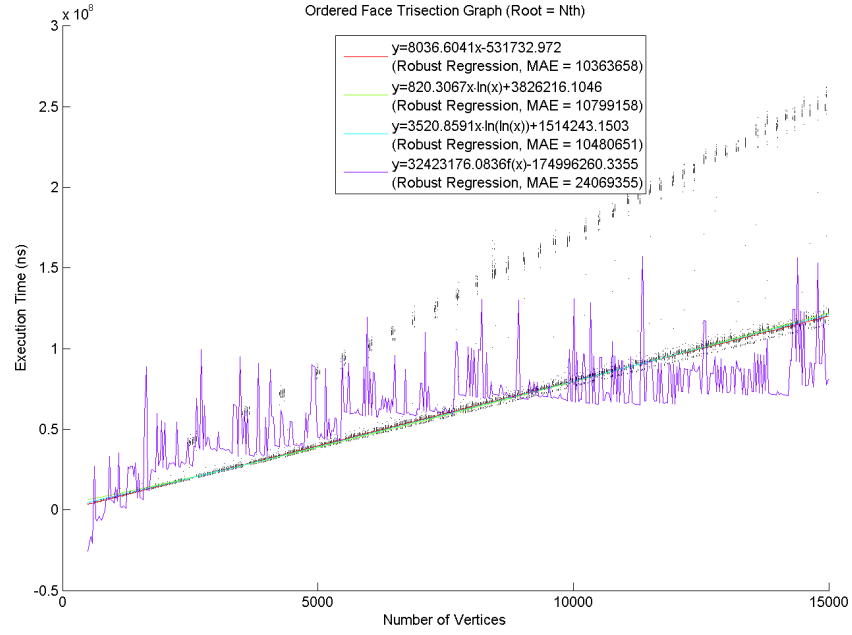
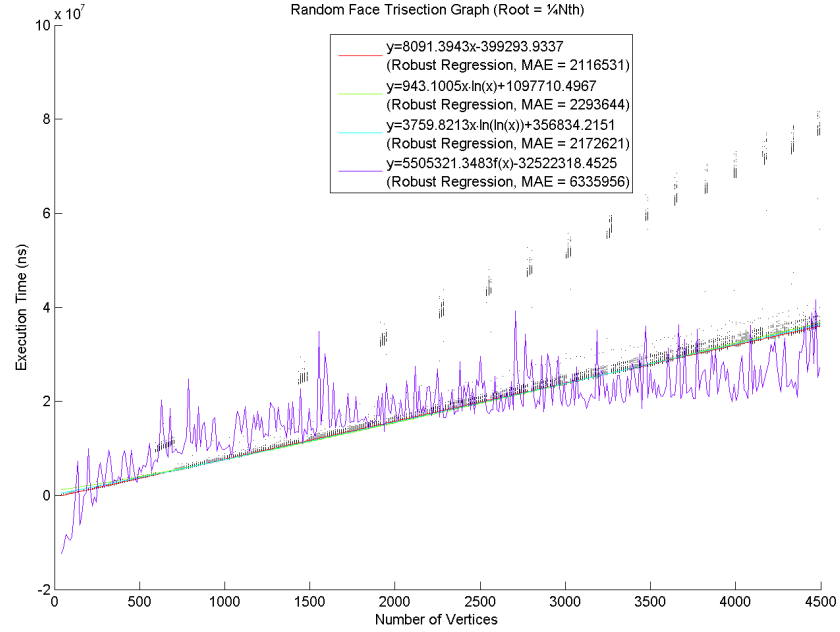


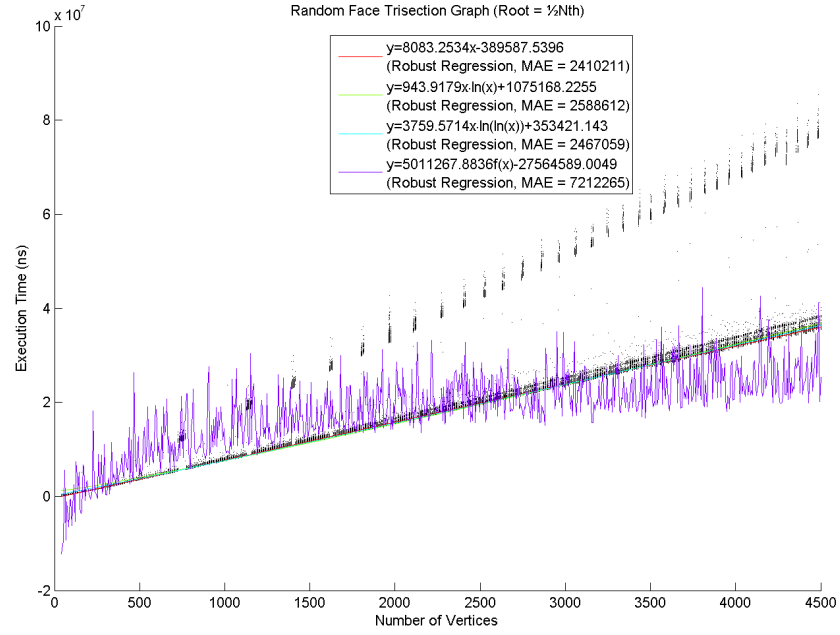
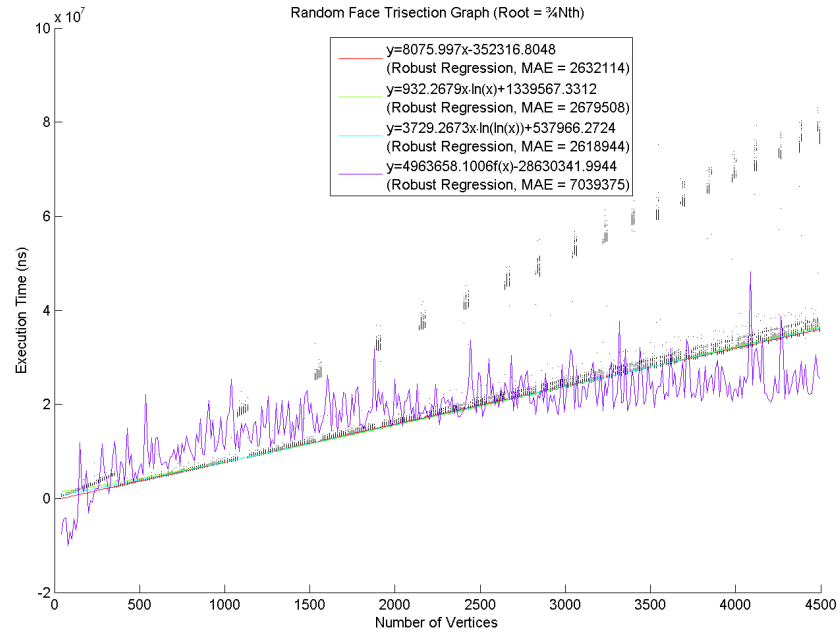
Figure A.2.31: Ordered Face Trisection Graph (500-15000 Vertices, $1/4N^{\text{th}}$ Vertex Root)

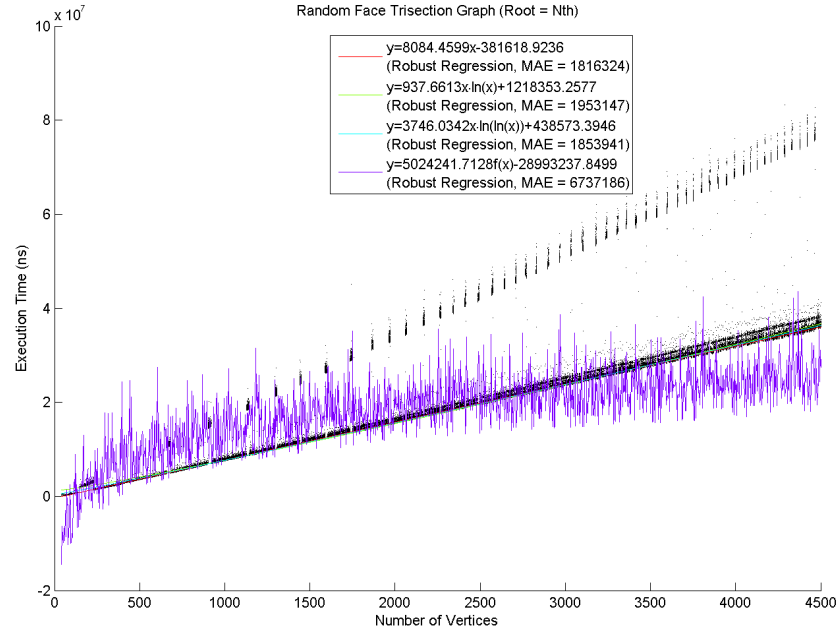
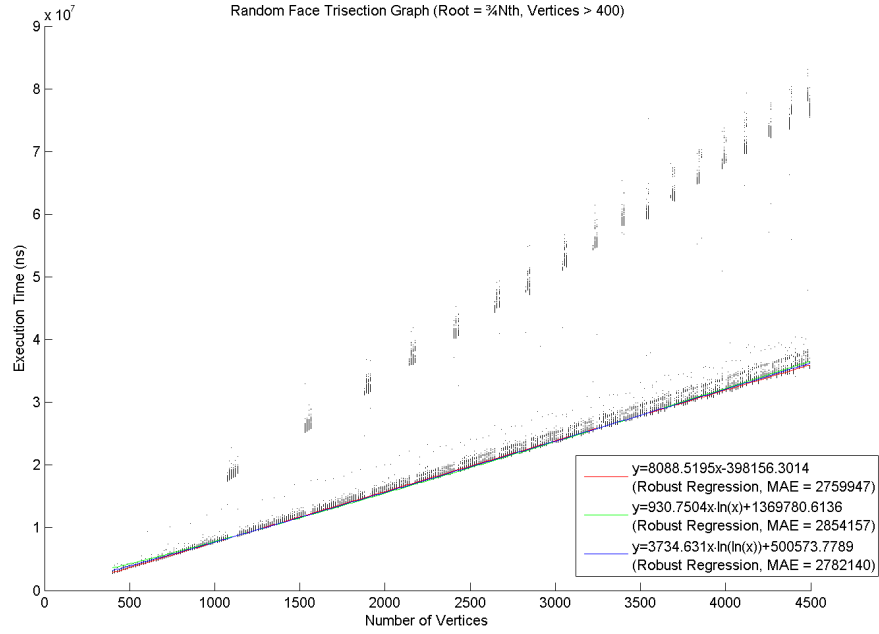
Figure A.2.32: Ordered Face Trisection Graph (500-15000 Vertices, $\frac{1}{2}N^{\text{th}}$ Vertex Root)Figure A.2.33: Ordered Face Trisection Graph (500-15000 Vertices, $\frac{3}{4}N^{\text{th}}$ Vertex Root)

Figure A.2.34: Ordered Face Trisection Graph (500-15000 Vertices, N^{th} Vertex Root)

A.2.7 Random Face Trisection Graphs (42-4500 Vertices)

Figure A.2.35: Random Face Trisection Graph (42-4500 Vertices, $\frac{1}{4}N^{\text{th}}$ Vertex Root)

Figure A.2.36: Random Face Trisection Graph (42-4500 Vertices, $\frac{1}{2}N^{\text{th}}$ Vertex Root)Figure A.2.37: Random Face Trisection Graph (42-4500 Vertices, $\frac{3}{4}N^{\text{th}}$ Vertex Root)

Figure A.2.38: Random Face Trisection Graph (42-4500 Vertices, N^{th} Vertex Root)Figure A.2.39: Random Face Trisection Graph (400-4500 Vertices, $\frac{3}{4}N^{\text{th}}$ Vertex Root)

A.2.8 Random Face Trisection Graphs (500-15000 Vertices)

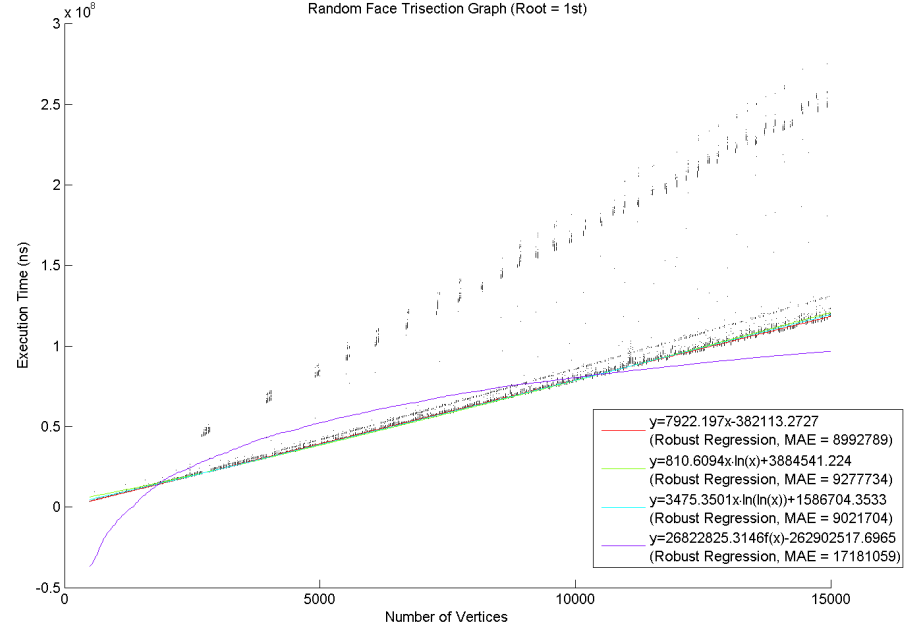


Figure A.2.40: Random Face Trisection Graph (500-15000 Vertices, 1st Vertex Root)

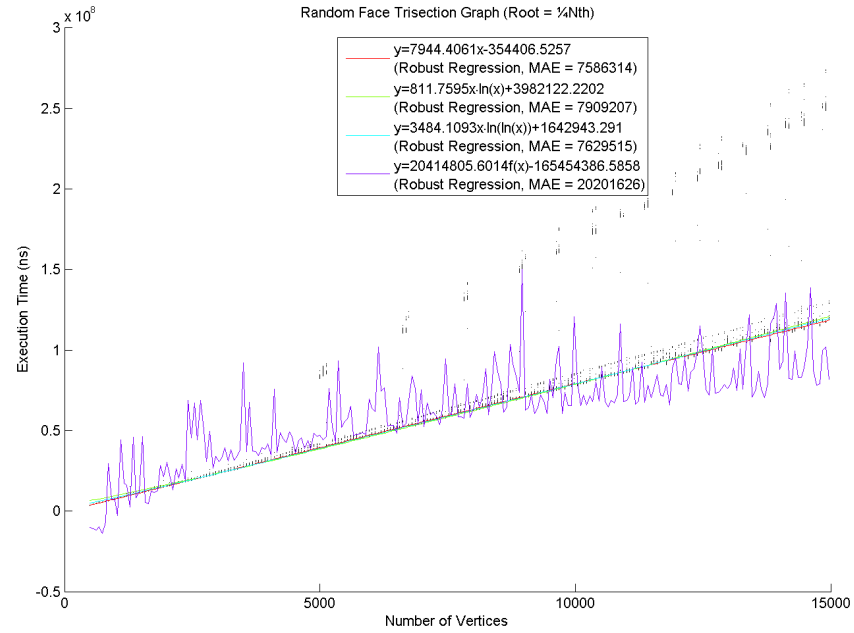
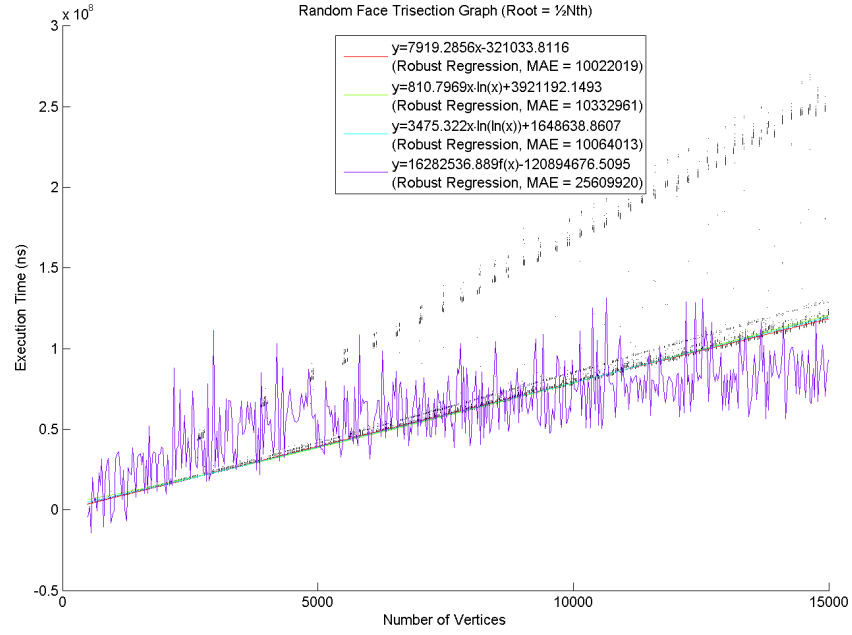
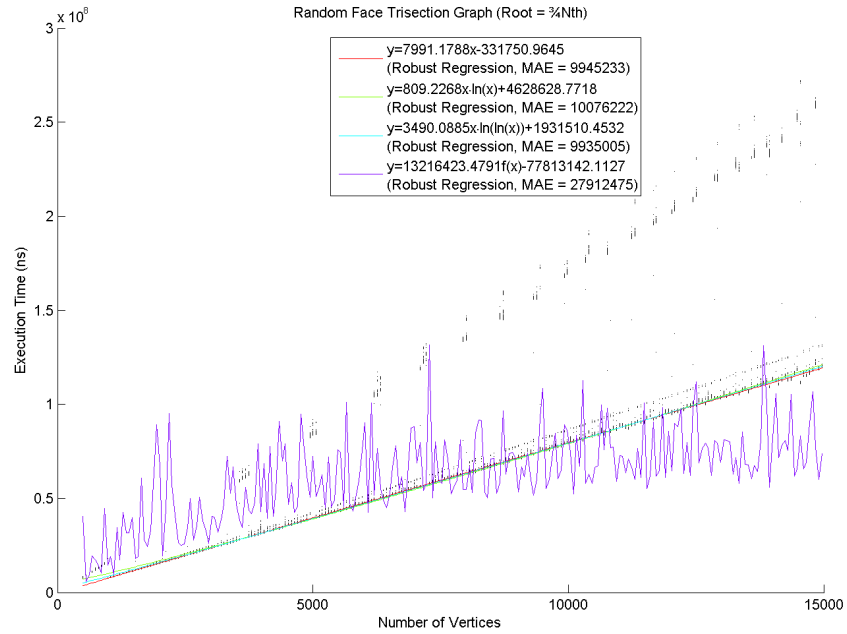
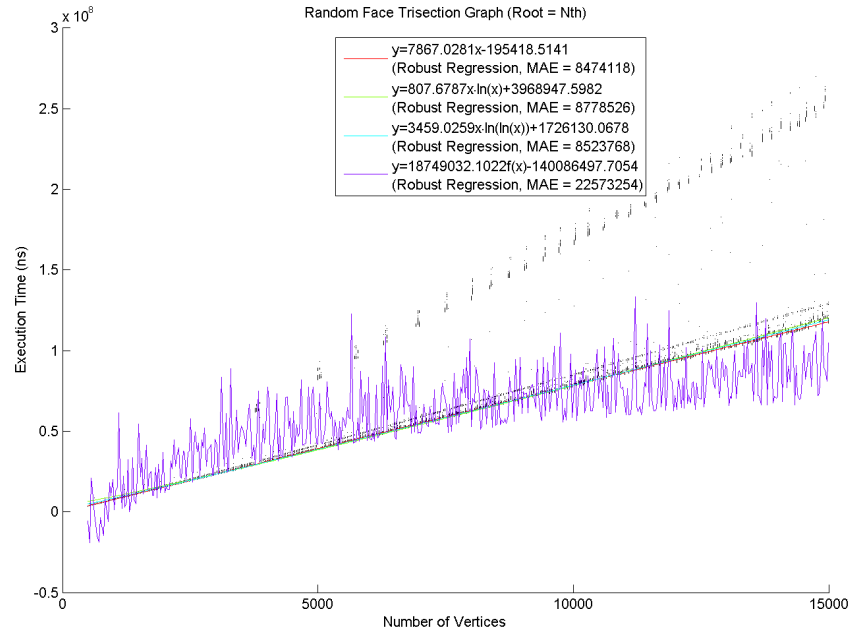
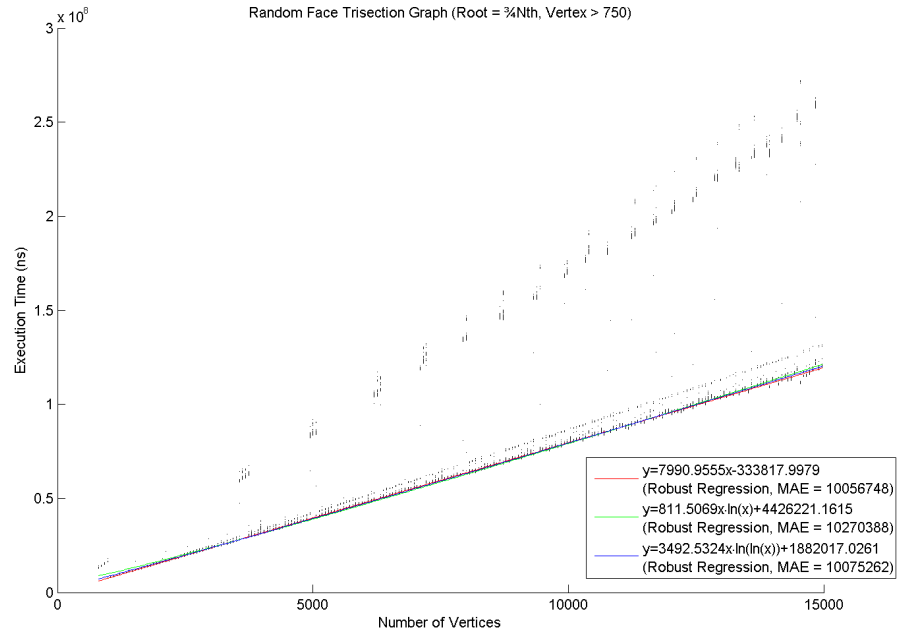


Figure A.2.41: Random Face Trisection Graph (500-15000 Vertices, $\frac{1}{4}N^{\text{th}}$ Vertex Root)

Figure A.2.42: Random Face Trisection Graph (500-15000 Vertices, $\frac{1}{2}N^{\text{th}}$ Vertex Root)Figure A.2.43: Random Face Trisection Graph (500-15000 Vertices, $\frac{3}{4}N^{\text{th}}$ Vertex Root)

Figure A.2.44: Random Face Trisection Graph (500-15000 Vertices, N^{th} Vertex Root)Figure A.2.45: Random Face Trisection Graph (750-15000 Vertices, $\frac{3}{4}N^{\text{th}}$ Vertex Root)

A.3 Variations in Number of Edges for Sub-Graphs of Constant Graph Size and Type

A.3.1 Triangular Prism Graphs (14999 Vertices)

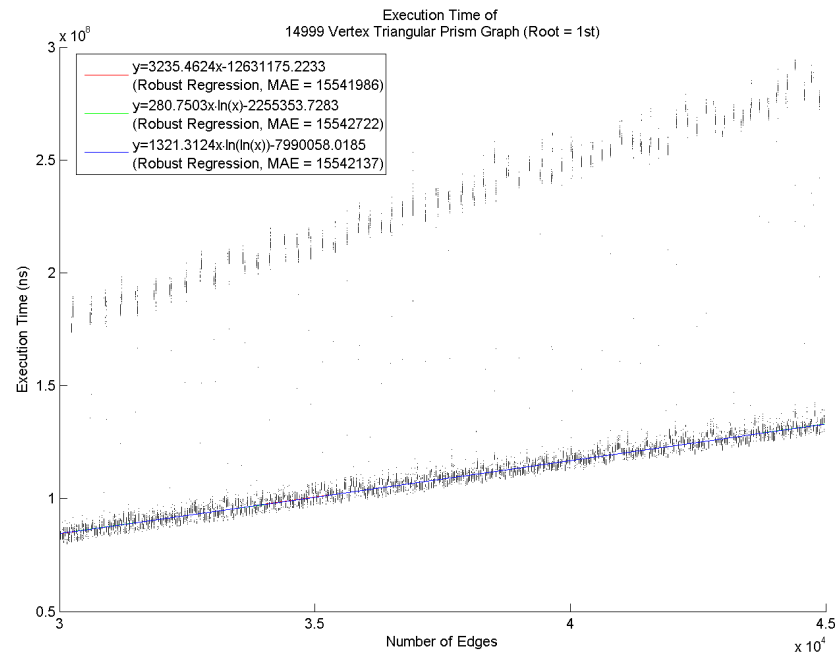


Figure A.3.1: Triangular Prism Graph (14999 Vertices, ~30000-45000 Edges, 1st Vertex Root)

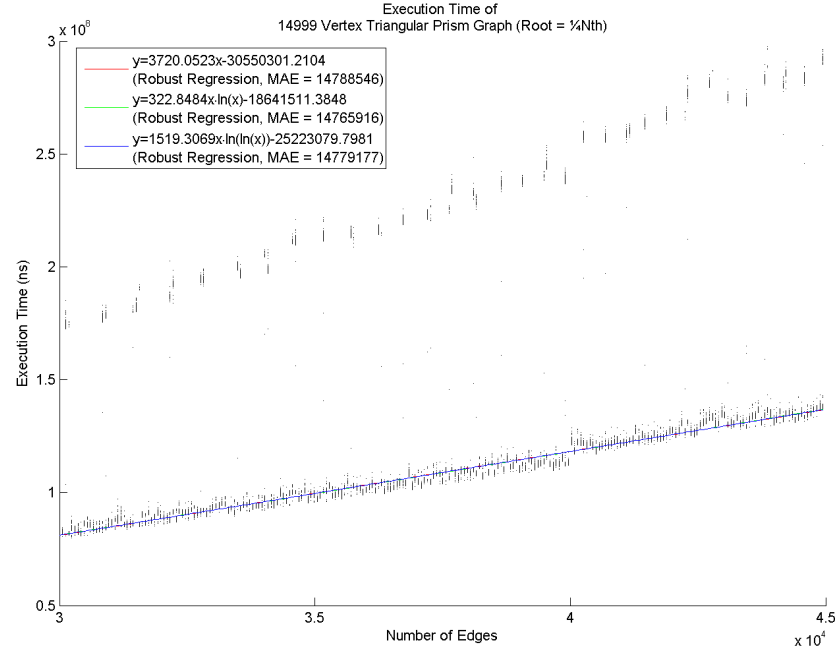


Figure A.3.2: Triangular Prism Graph (14999 Vertices, ~ 30000 -45000 Edges, $\frac{1}{4}N^{\text{th}}$ Vertex Root)

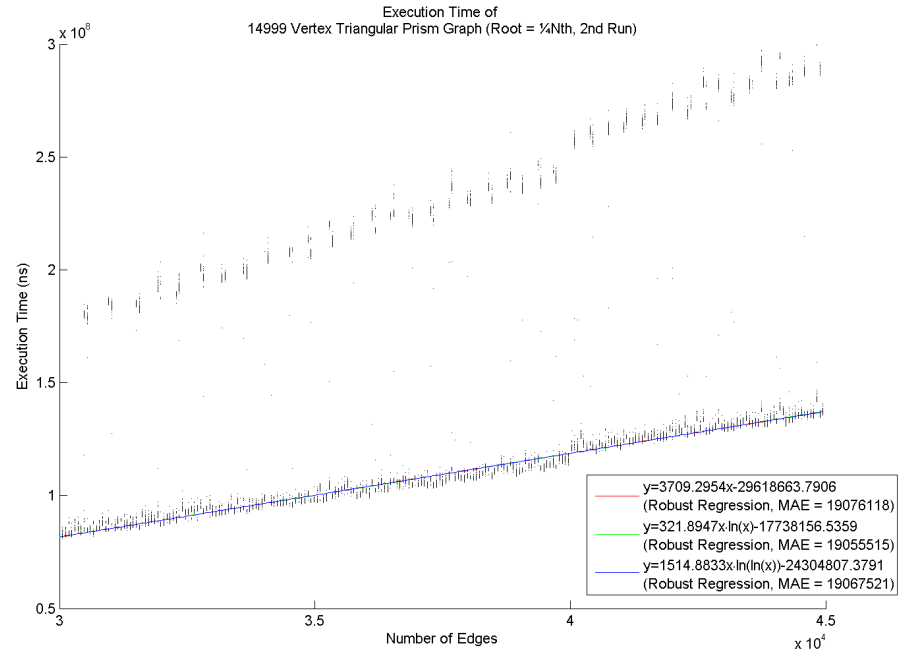


Figure A.3.3: Triangular Prism Graph (14999 Vertices, ~ 30000 -45000 Edges, $\frac{1}{4}N^{\text{th}}$ Vertex Root, 2nd Run)

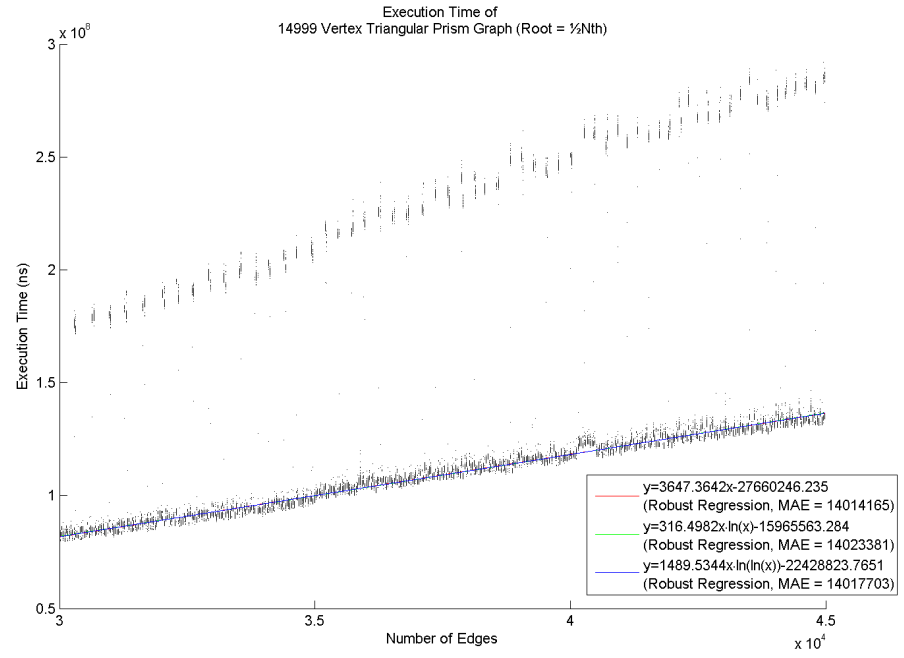


Figure A.3.4: Triangular Prism Graph (14999 Vertices, ~ 30000 -45000 Edges, $\frac{1}{2}N^{\text{th}}$ Vertex Root)

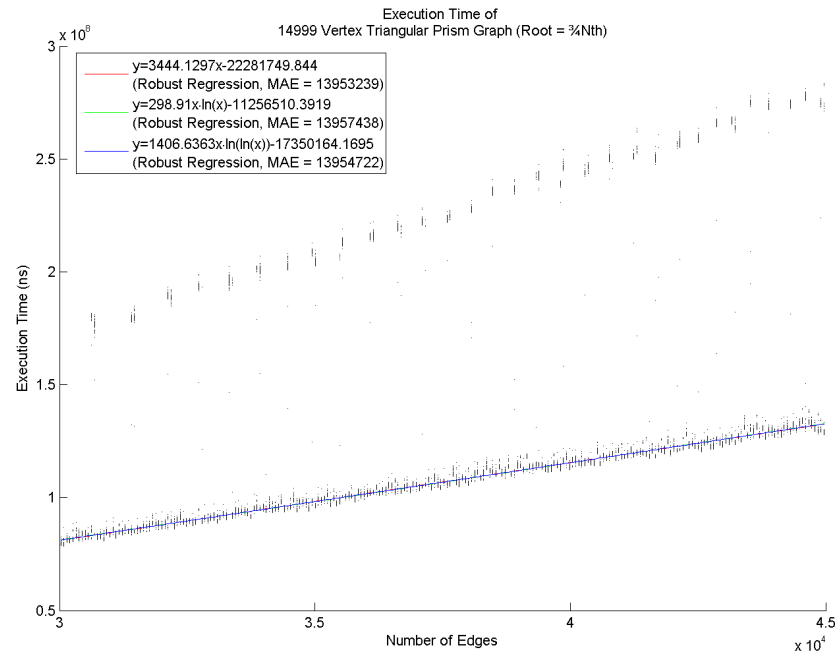


Figure A.3.5: Triangular Prism Graph (14999 Vertices, ~ 30000 -45000 Edges, $\frac{3}{4}N^{\text{th}}$ Vertex Root)

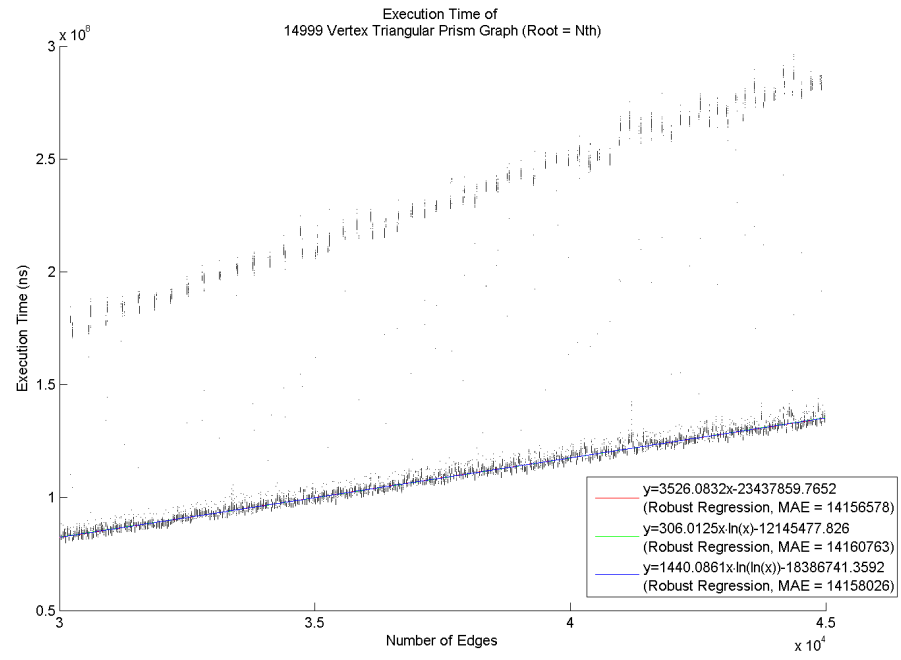
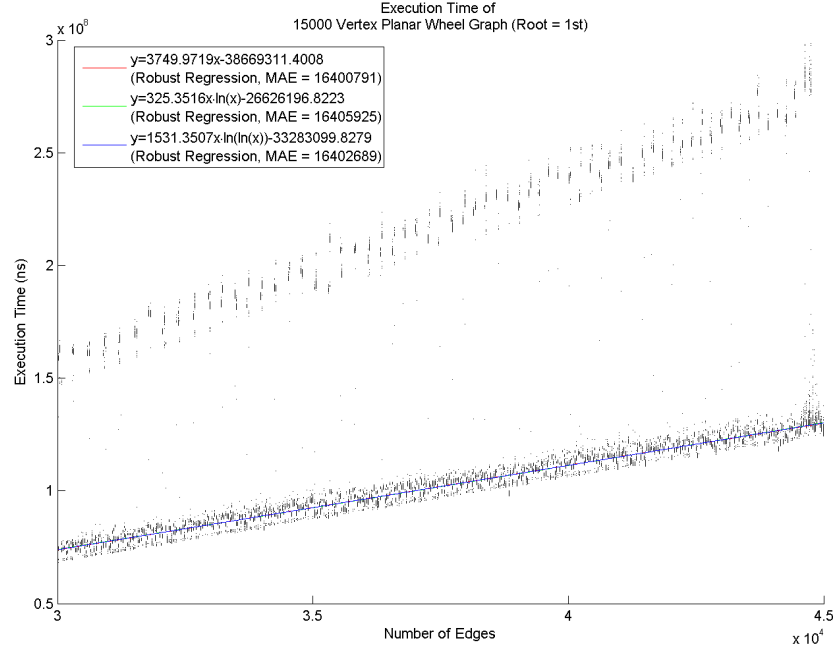
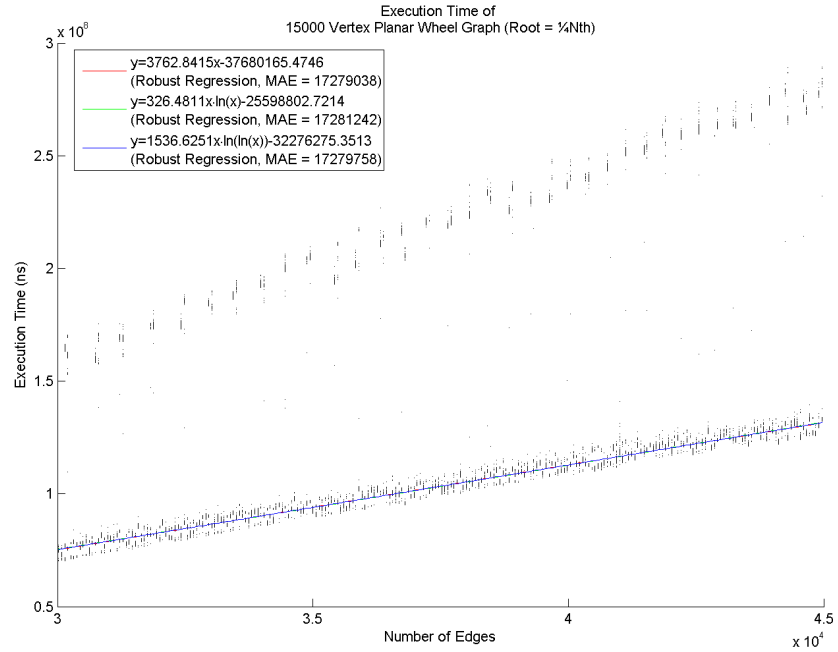
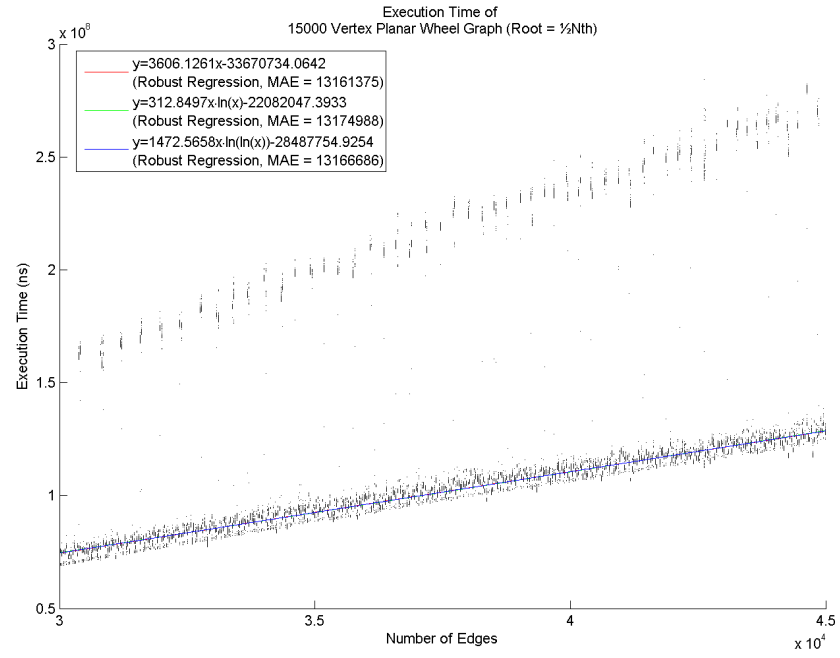
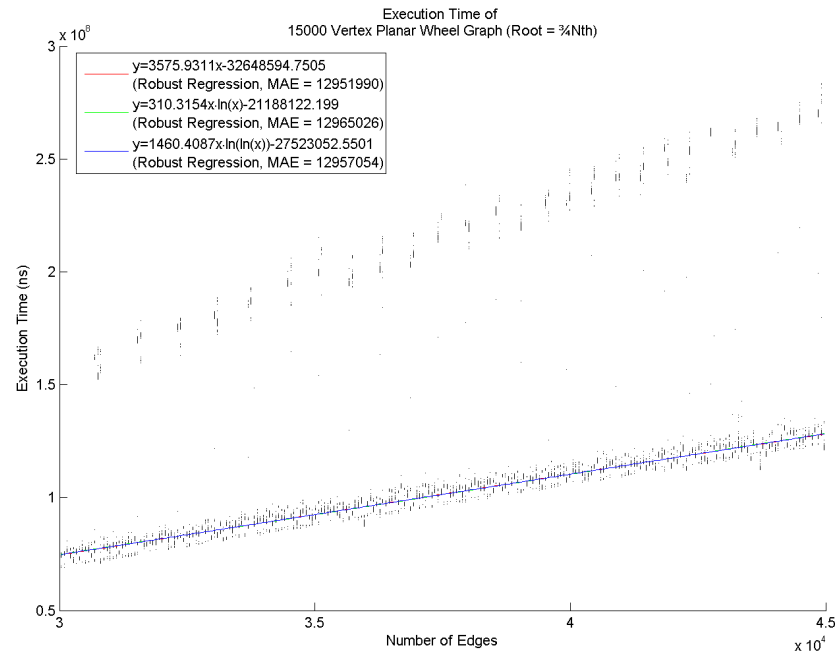
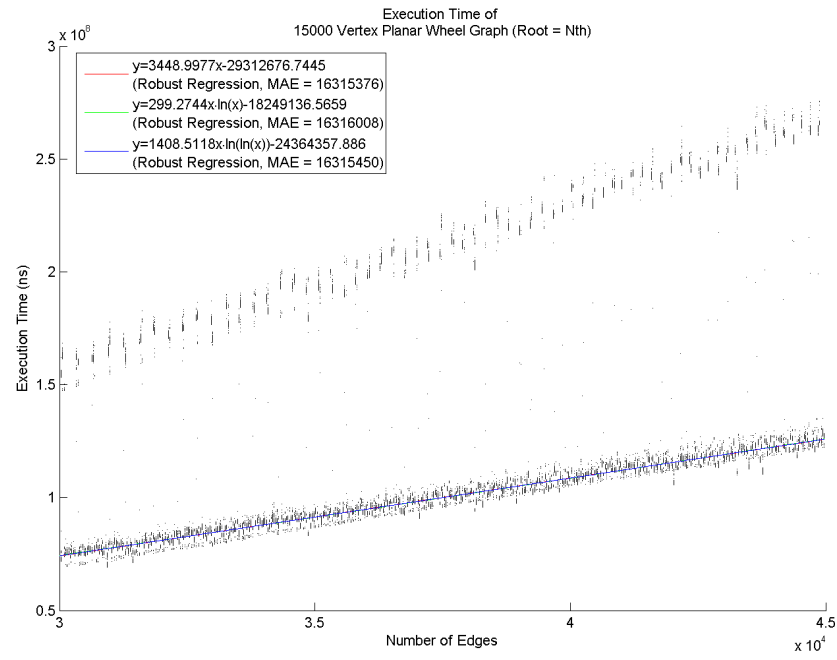


Figure A.3.6: Triangular Prism Graph (14999 Vertices, ~ 30000 -45000 Edges, N^{th} Vertex Root)

A.3.2 Planar Wheel Graphs (15000 Vertices)

Figure A.3.7: Planar Wheel Graph (15000 Vertices, ~ 30000 -45000 Edges, 1st Vertex Root)Figure A.3.8: Planar Wheel Graph (15000 Vertices, ~ 30000 -45000 Edges, $\frac{1}{4}N^{\text{th}}$ Vertex Root)

Figure A.3.9: Planar Wheel Graph (15000 Vertices, ~ 30000 -45000 Edges, $\frac{1}{2}N^{\text{th}}$ Vertex Root)Figure A.3.10: Planar Wheel Graph (15000 Vertices, ~ 30000 -45000 Edges, $\frac{3}{4}N^{\text{th}}$ Vertex Root)

Figure A.3.11: Planar Wheel Graph (15000 Vertices, ~ 30000 -45000 Edges, N^{th} Vertex Root)

A.3.3 Ordered Face Trisection Graphs (14999 Vertices)

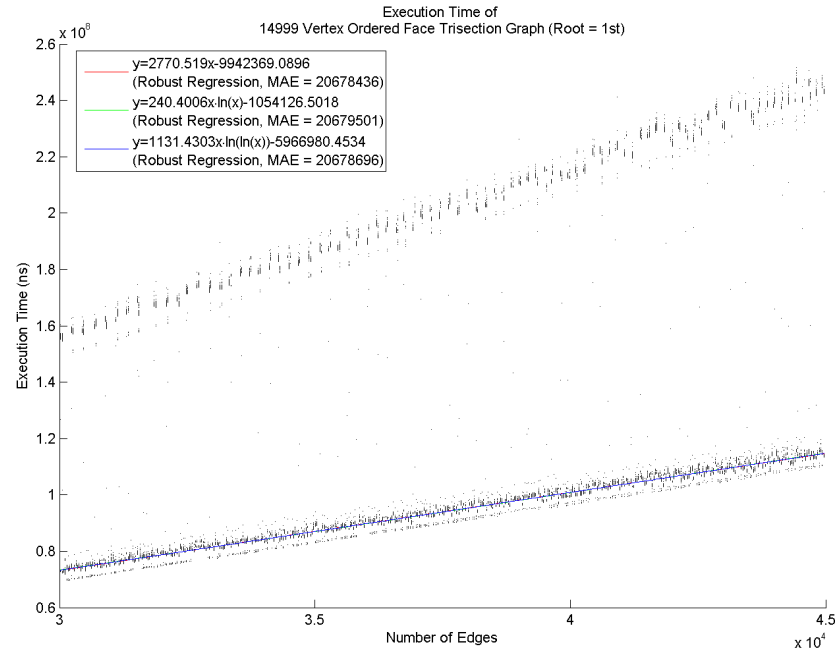


Figure A.3.12: Ordered Face Trisection Graph (14999 Vertices, ~30000-45000 Edges, 1st Vertex Root)

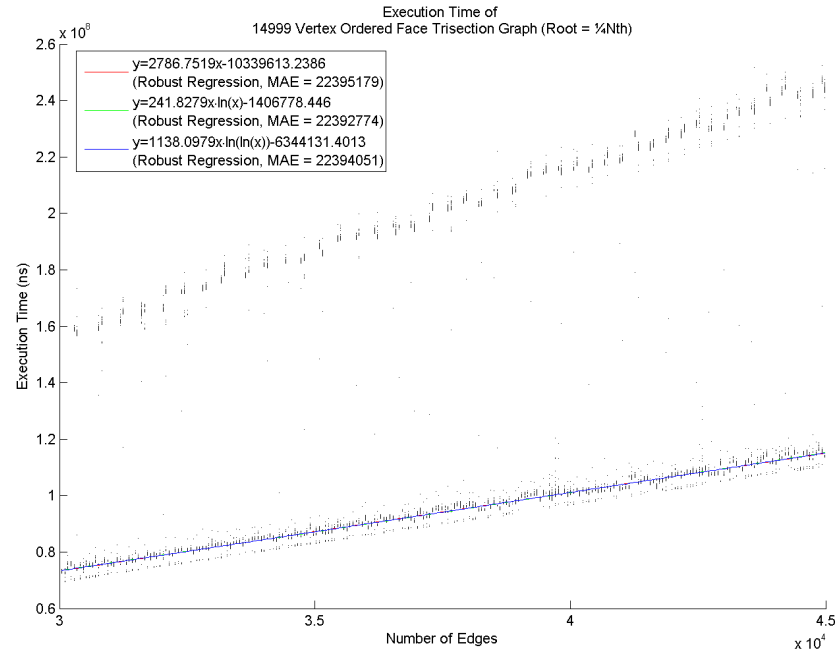


Figure A.3.13: Ordered Face Trisection Graph (14999 Vertices, ~ 30000 -45000 Edges, $\frac{1}{4}N^{\text{th}}$ Vertex Root)

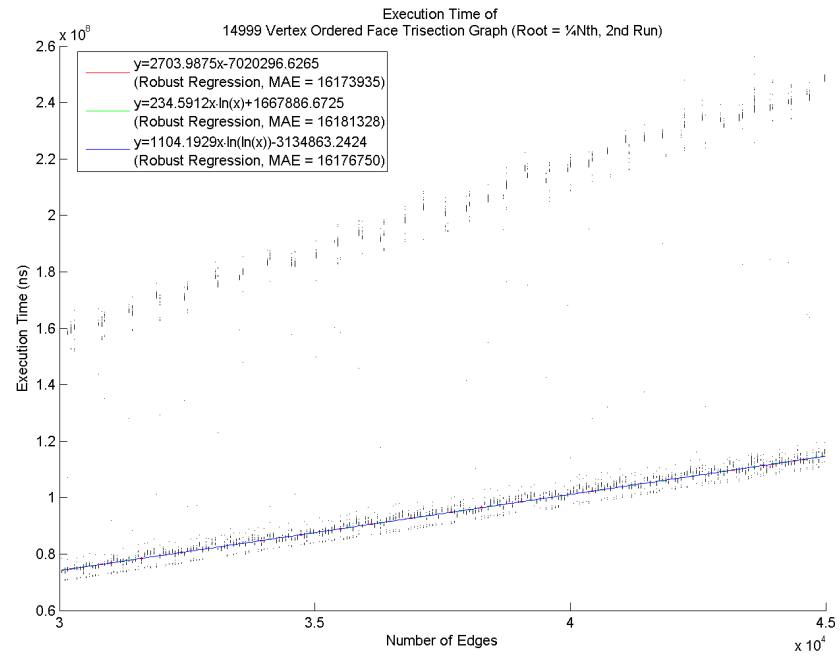


Figure A.3.14: Ordered Face Trisection Graph (14999 Vertices, ~ 30000 -45000 Edges, $\frac{1}{4}N^{\text{th}}$ Vertex Root, 2nd Run)

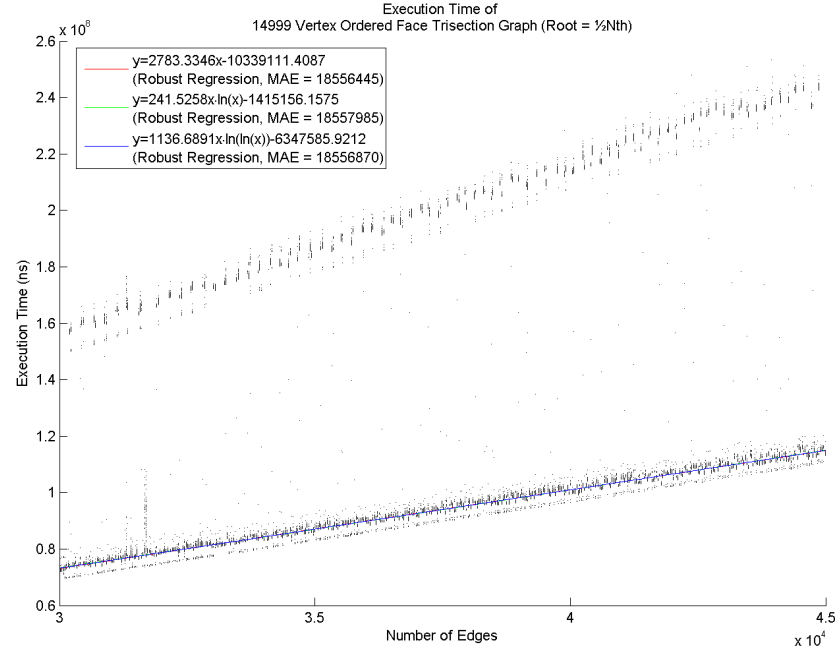


Figure A.3.15: Ordered Face Trisection Graph (14999 Vertices, ~ 30000 -45000 Edges, $\frac{1}{2}N^{\text{th}}$ Vertex Root)

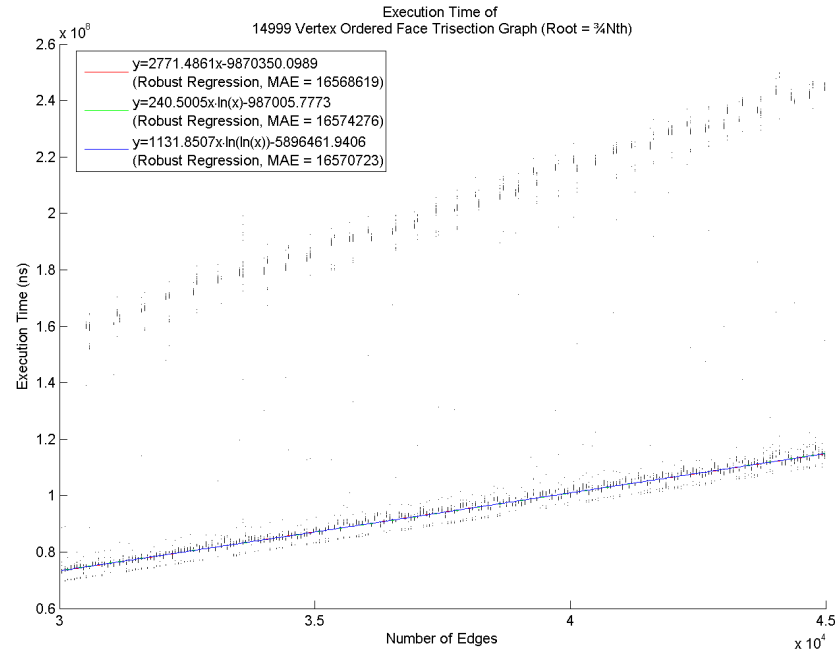


Figure A.3.16: Ordered Face Trisection Graph (14999 Vertices, ~ 30000 -45000 Edges, $\frac{3}{4}N^{\text{th}}$ Vertex Root)

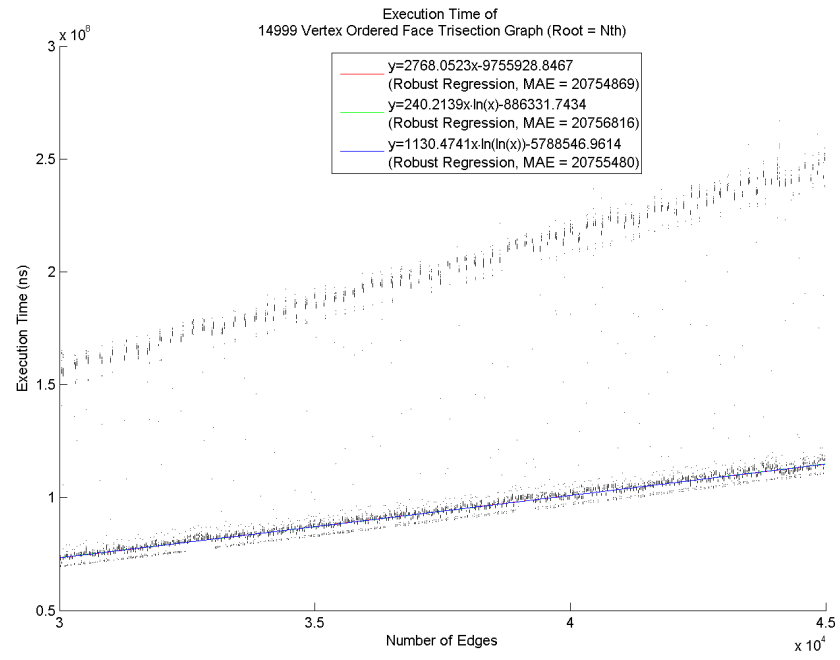


Figure A.3.17: Ordered Face Trisection Graph (14999 Vertices, ~ 30000 -45000 Edges, N^{th} Vertex Root)

A.3.4 Random Face Trisection Graphs (14999 Vertices)

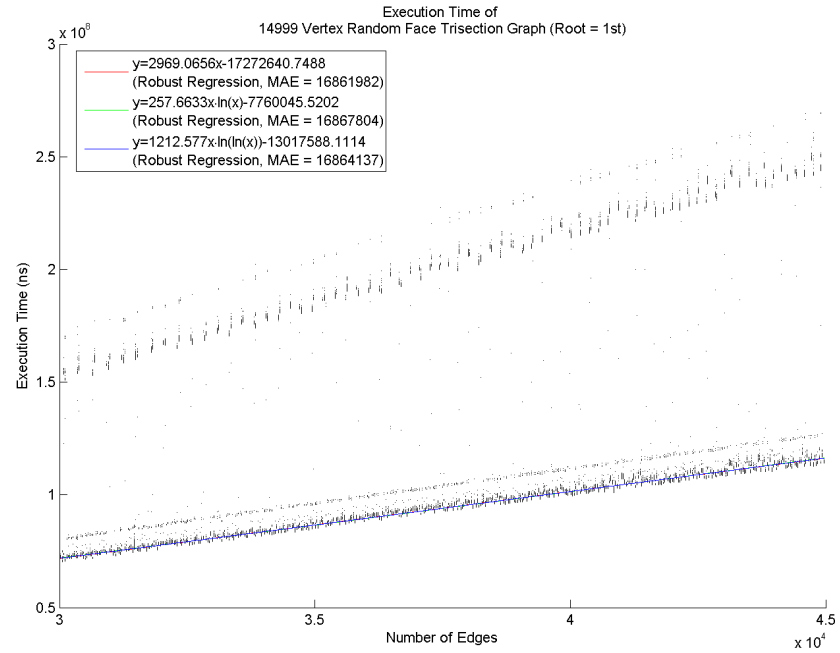


Figure A.3.18: Random Face Trisection Graph (14999 Vertices, ~ 30000 -45000 Edges, 1st Vertex Root)

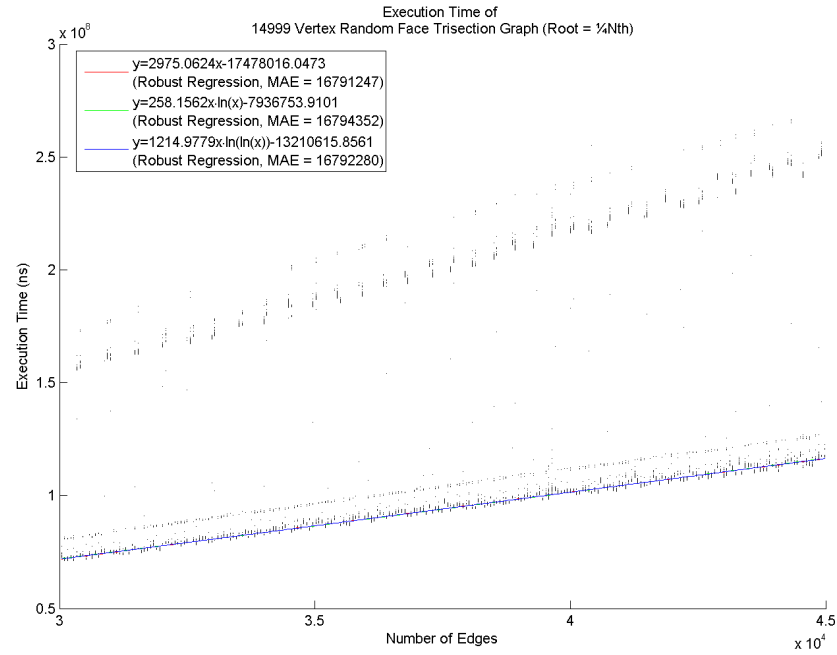


Figure A.3.19: Random Face Trisection Graph (14999 Vertices, ~ 30000 -45000 Edges, $\frac{1}{4}N^{\text{th}}$ Vertex Root)

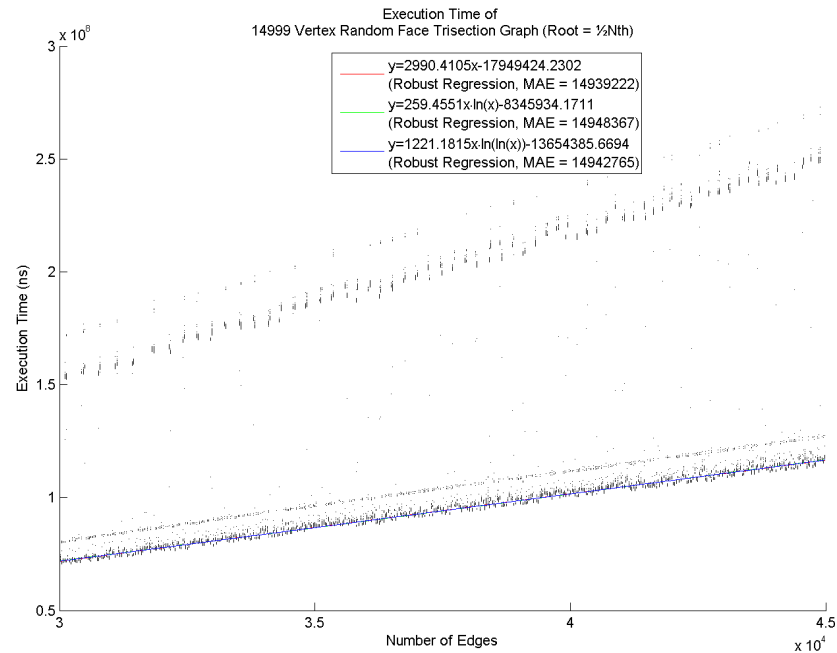


Figure A.3.20: Random Face Trisection Graph (14999 Vertices, ~ 30000 -45000 Edges, $\frac{1}{2}N^{\text{th}}$ Vertex Root)

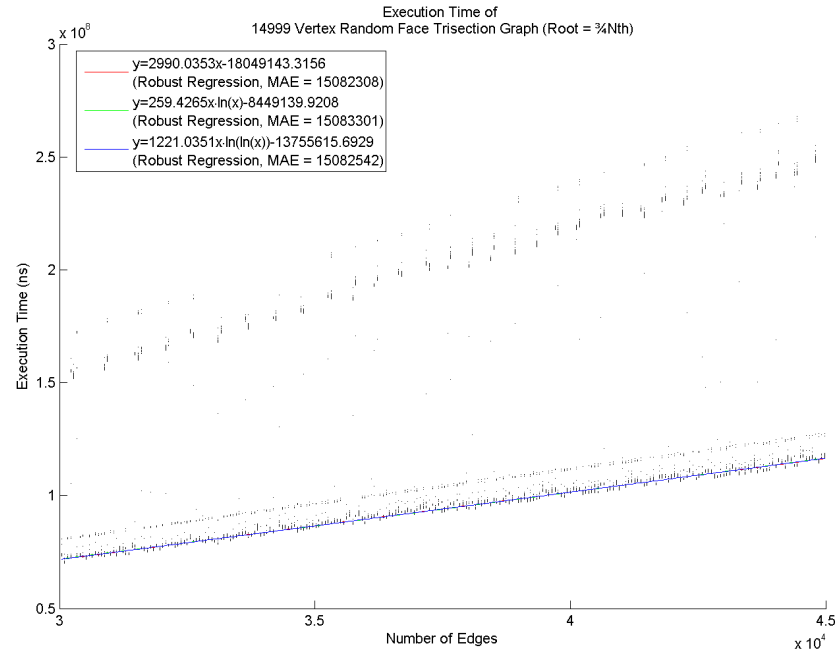


Figure A.3.21: Random Face Trisection Graph (14999 Vertices, ~ 30000 -45000 Edges, $\frac{3}{4}N^{\text{th}}$ Vertex Root)

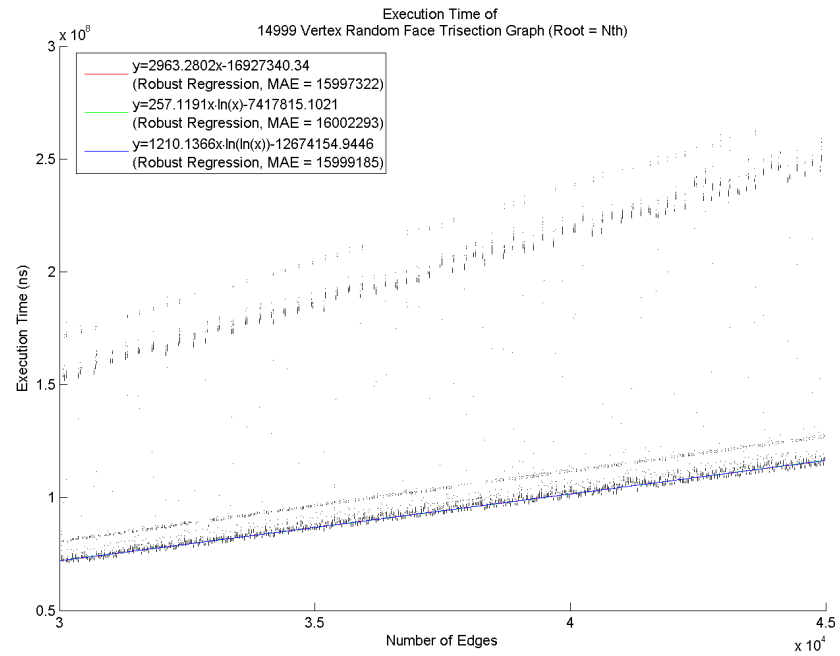


Figure A.3.22: Random Face Trisection Graph (14999 Vertices, ~ 30000 -45000 Edges, N^{th} Vertex Root)

A.4 Variations in Number of Edges for Sub-Graphs of Constant Graph Size and Type – Effects on Numbers of Conflicts and Mean Segment Depth

A.4.1 Triangular Prism Graphs

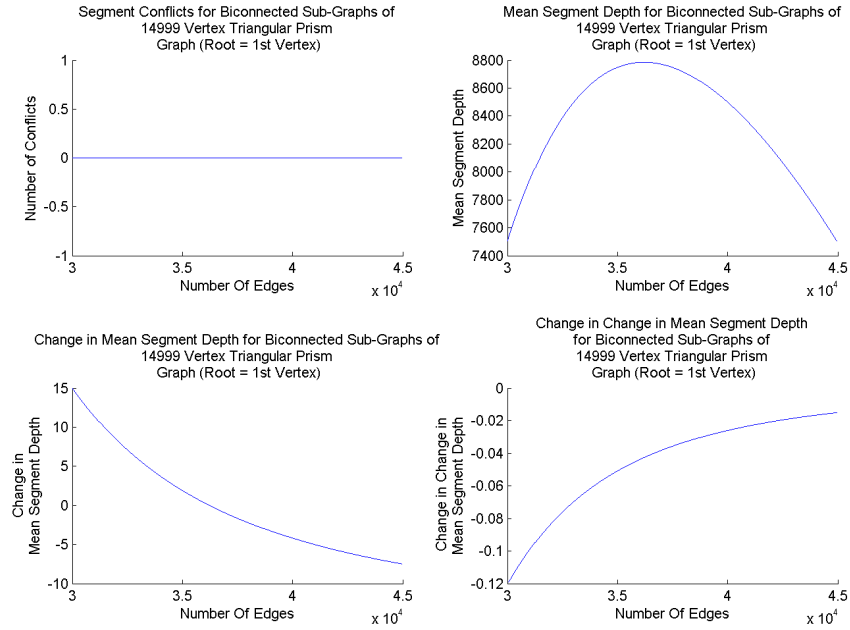


Figure A.4.1: Mean Segment Depth for Biconnected Sub-Graphs of Triangular Prism Graph (14999 Vertices, ~ 30000 - 45000 Edges, 1st Vertex Root)

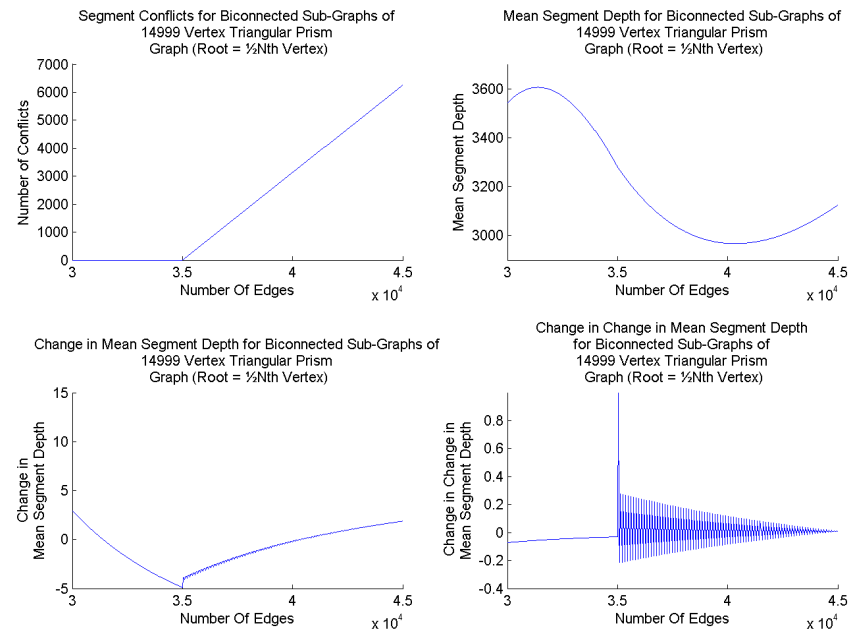


Figure A.4.2: Mean Segment Depth for Biconnected Sub-Graphs of Triangular Prism Graph (14999 Vertices, ~ 30000 - 45000 Edges, $1/2N^{\text{th}}$ Vertex Root)

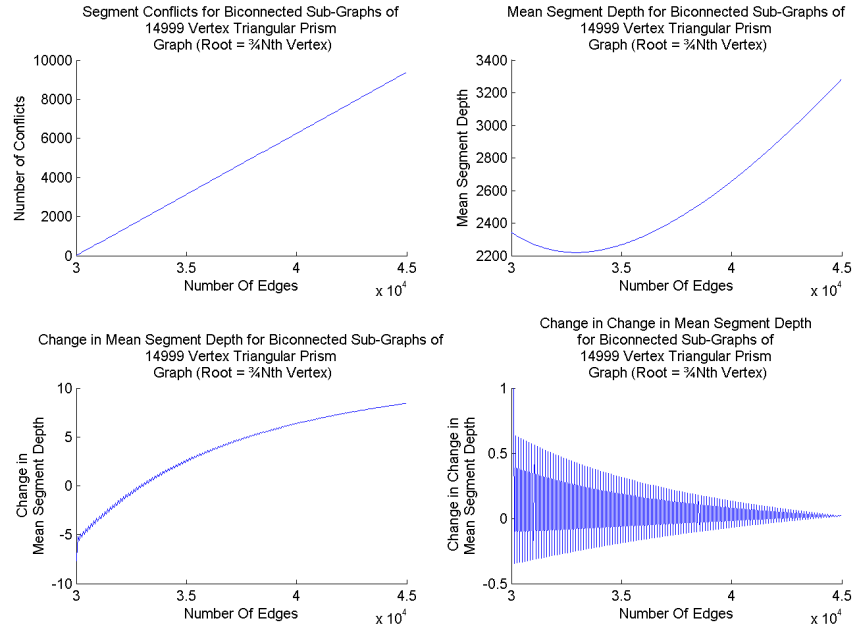


Figure A.4.3: Mean Segment Depth for Biconnected Sub-Graphs of Triangular Prism Graph (14999 Vertices, ~ 30000 - 45000 Edges, $\frac{3}{4}N^{\text{th}}$ Vertex Root)

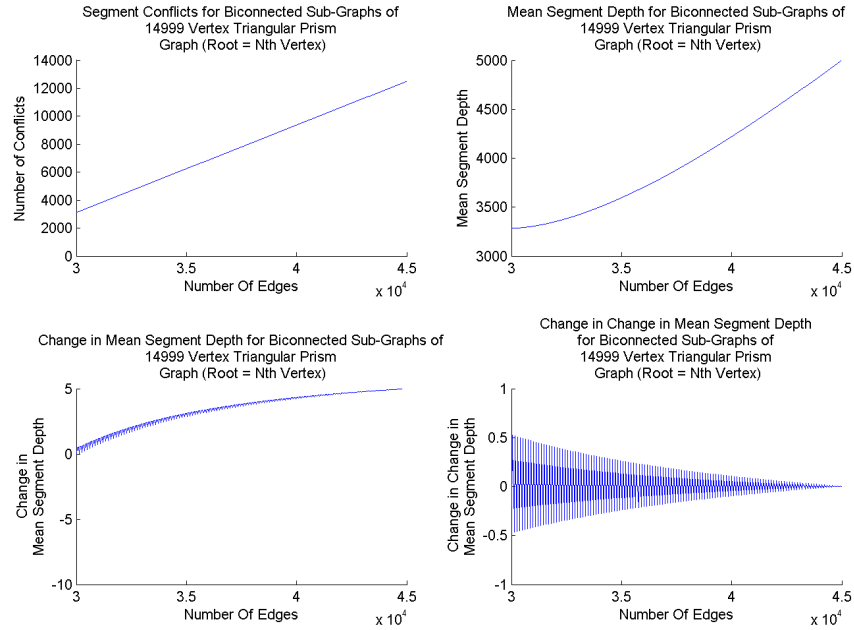


Figure A.4.4: Mean Segment Depth for Biconnected Sub-Graphs of Triangular Prism Graph (14999 Vertices, ~ 30000 - 45000 Edges, N^{th} Vertex Root)

A.4.2 Planar Wheel Graphs

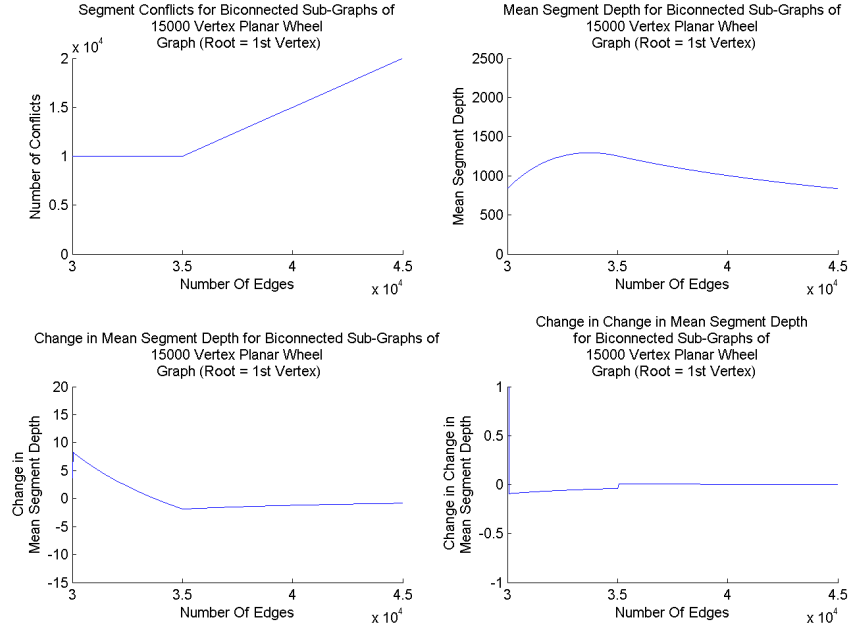


Figure A.4.5: Mean Segment Depth for Biconnected Sub-Graphs of Planar Wheel Graph (14999 Vertices, ~ 30000 - 45000 Edges, 1st Vertex Root)

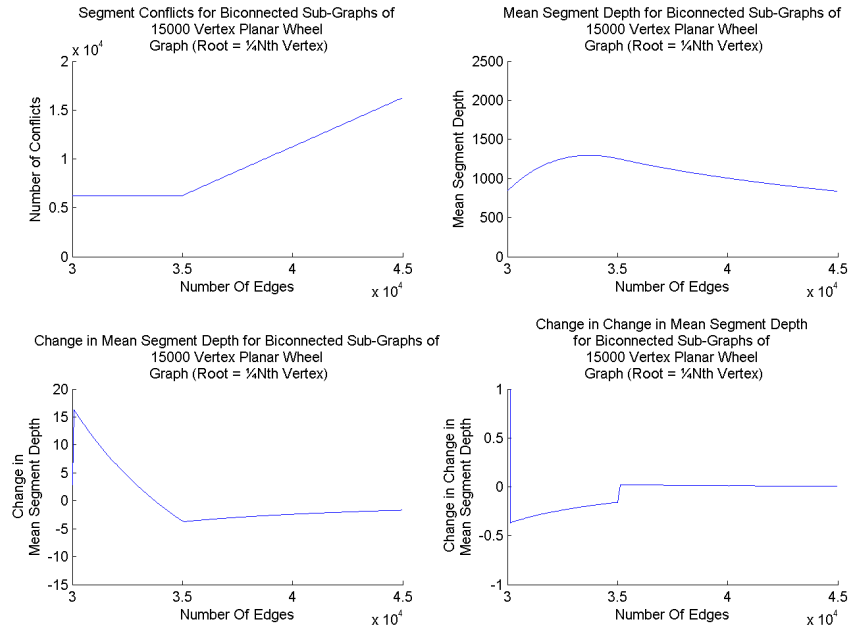


Figure A.4.6: Mean Segment Depth for Biconnected Sub-Graphs of Planar Wheel Graph (14999 Vertices, ~ 30000 - 45000 Edges, $\frac{1}{4}N^{\text{th}}$ Vertex Root)

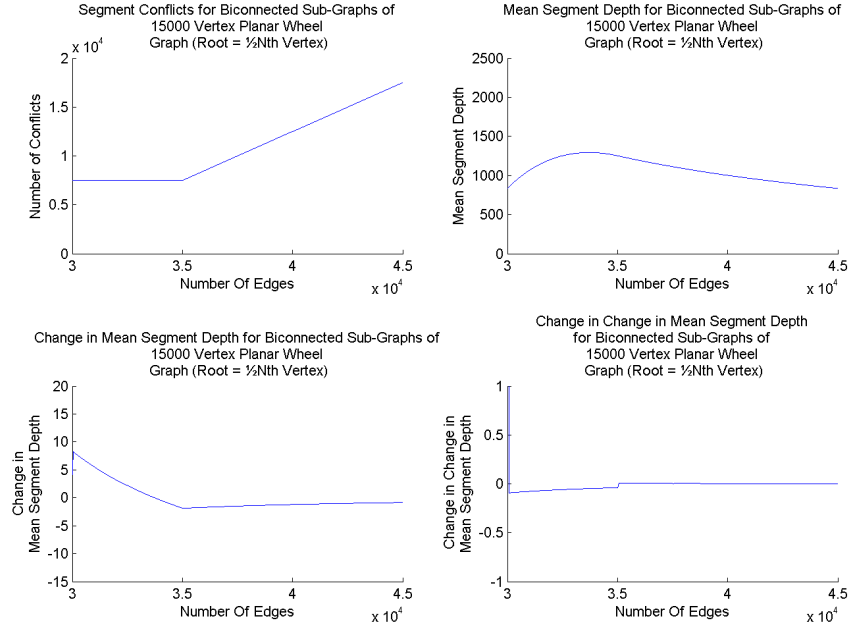


Figure A.4.7: Mean Segment Depth for Biconnected Sub-Graphs of Planar Wheel Graph (14999 Vertices, ~ 30000 - 45000 Edges, $\frac{1}{2}N^{\text{th}}$ Vertex Root)

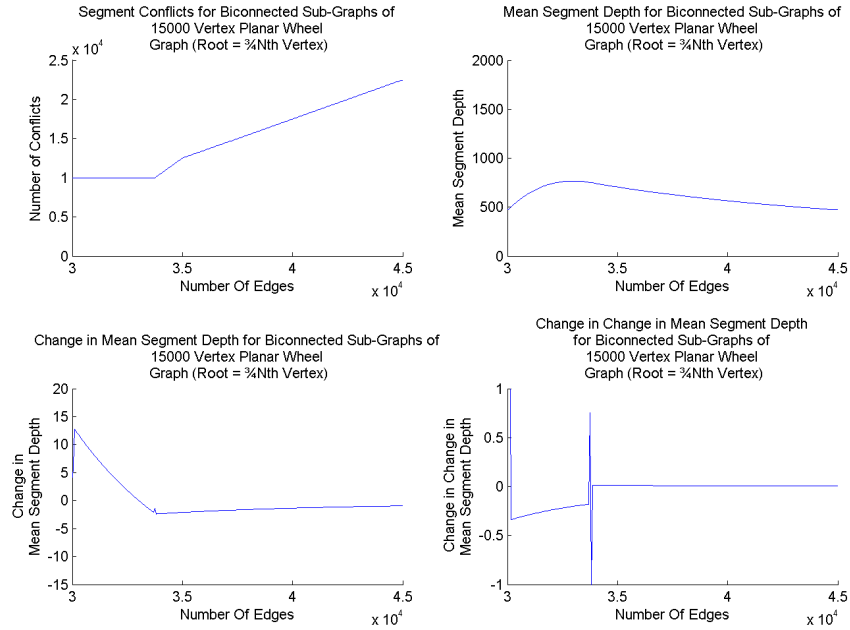


Figure A.4.8: Mean Segment Depth for Biconnected Sub-Graphs of Planar Wheel Graph (14999 Vertices, ~ 30000 - 45000 Edges, $\frac{3}{4}N^{\text{th}}$ Vertex Root)

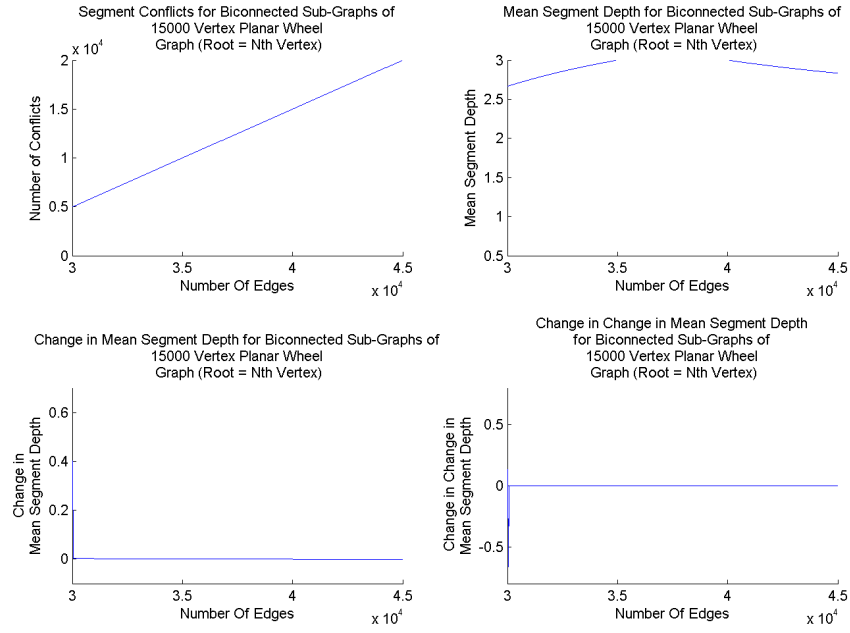


Figure A.4.9: Mean Segment Depth for Biconnected Sub-Graphs of Planar Wheel Graph (14999 Vertices, ~ 30000 -45000 Edges, N^{th} Vertex Root)

A.4.3 Ordered Face Trisection Graphs

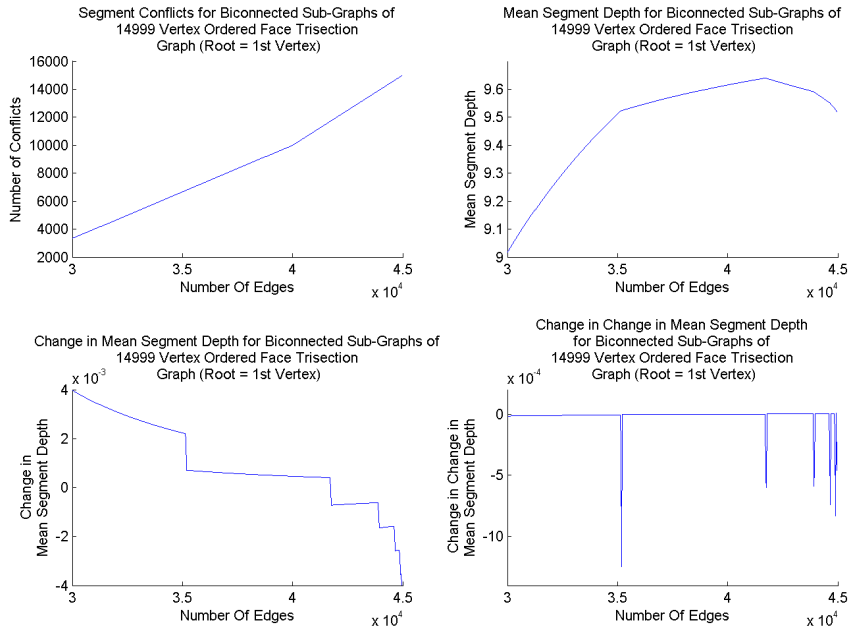


Figure A.4.10: Mean Segment Depth for Biconnected Sub-Graphs of Ordered Face Trisection Graph (14999 Vertices, ~ 30000 -45000 Edges, 1st Vertex Root)

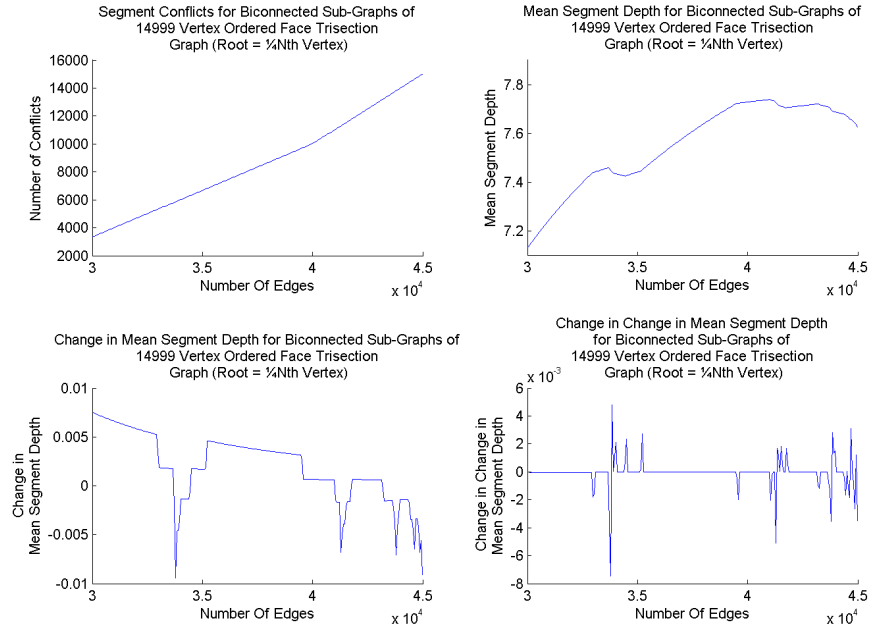


Figure A.4.11: Mean Segment Depth for Biconnected Sub-Graphs of Ordered Face Trisection Graph (14999 Vertices, ~ 30000 - 45000 Edges, $\frac{1}{4}N^{\text{th}}$ Vertex Root)

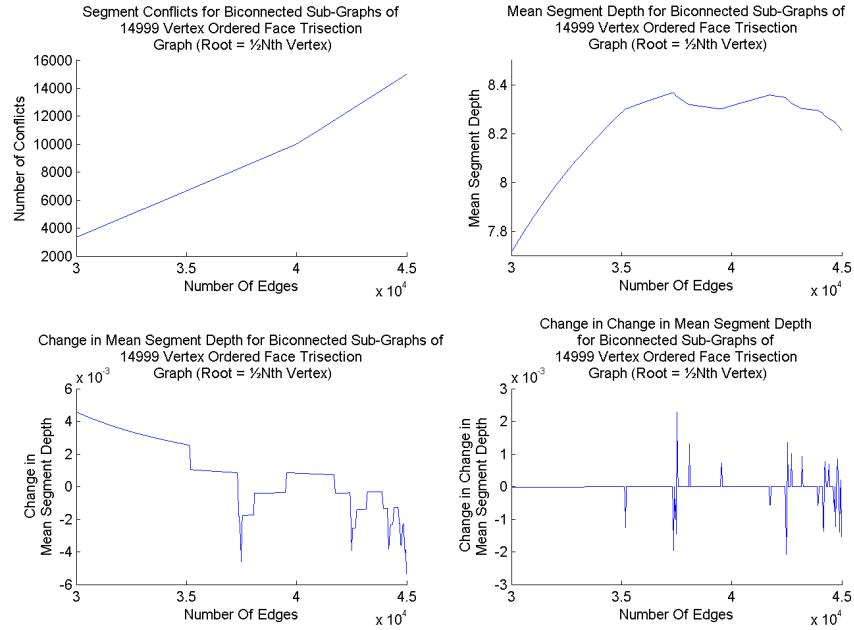


Figure A.4.12: Mean Segment Depth for Biconnected Sub-Graphs of Ordered Face Trisection Graph (14999 Vertices, ~ 30000 - 45000 Edges, $\frac{1}{2}N^{\text{th}}$ Vertex Root)

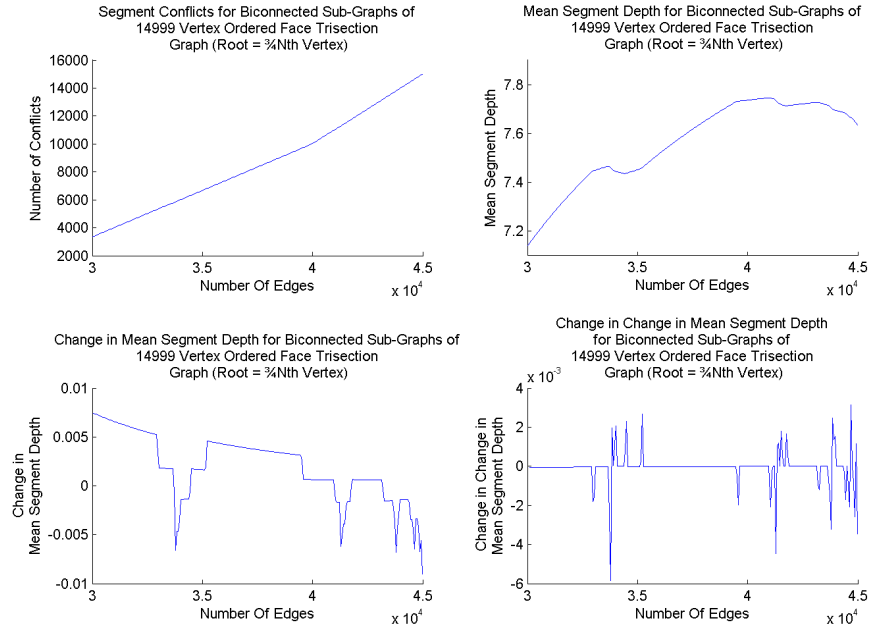


Figure A.4.13: Mean Segment Depth for Biconnected Sub-Graphs of Ordered Face Trisection Graph (14999 Vertices, ~ 30000 - 45000 Edges, $\frac{3}{4}N^{\text{th}}$ Vertex Root)

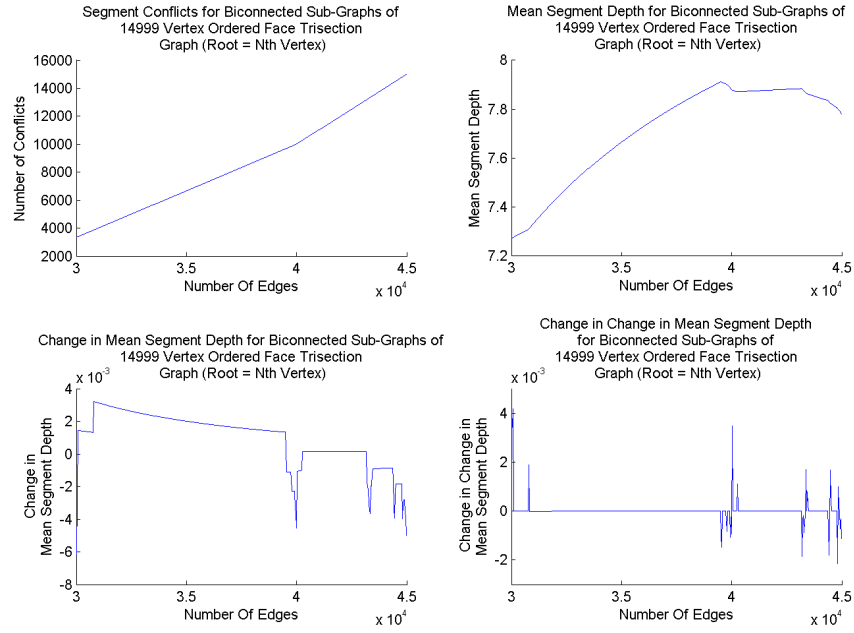


Figure A.4.14: Mean Segment Depth for Biconnected Sub-Graphs of Ordered Face Trisection Graph (14999 Vertices, ~ 30000 - 45000 Edges, N^{th} Vertex Root)

A.4.4 Random Face Trisection Graphs

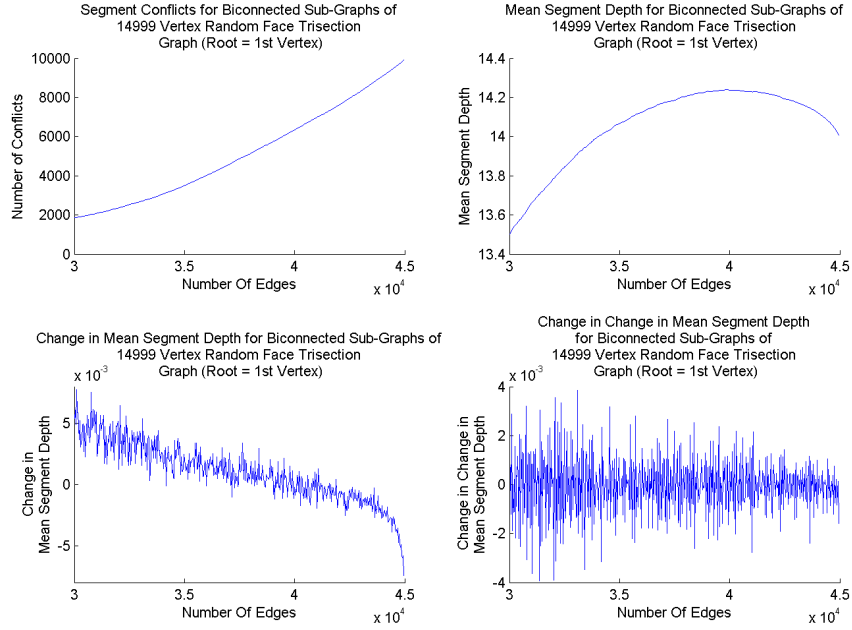


Figure A.4.15: Mean Segment Depth for Biconnected Sub-Graphs of Random Face Trisection Graph (14999 Vertices, ~ 30000 - 45000 Edges, 1st Vertex Root)

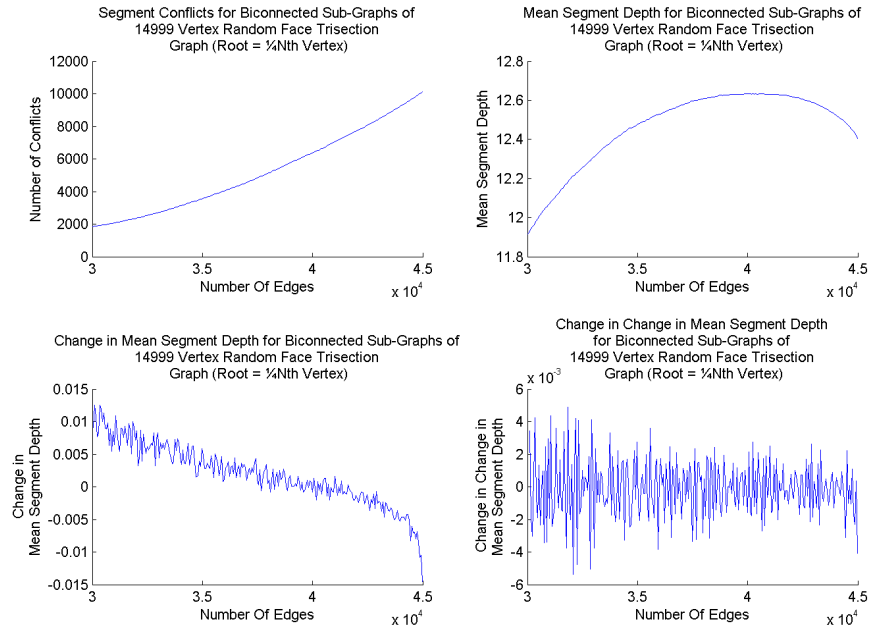


Figure A.4.16: Mean Segment Depth for Biconnected Sub-Graphs of Random Face Trisection Graph (14999 Vertices, ~ 30000 - 45000 Edges, $1/4N^{\text{th}}$ Vertex Root)

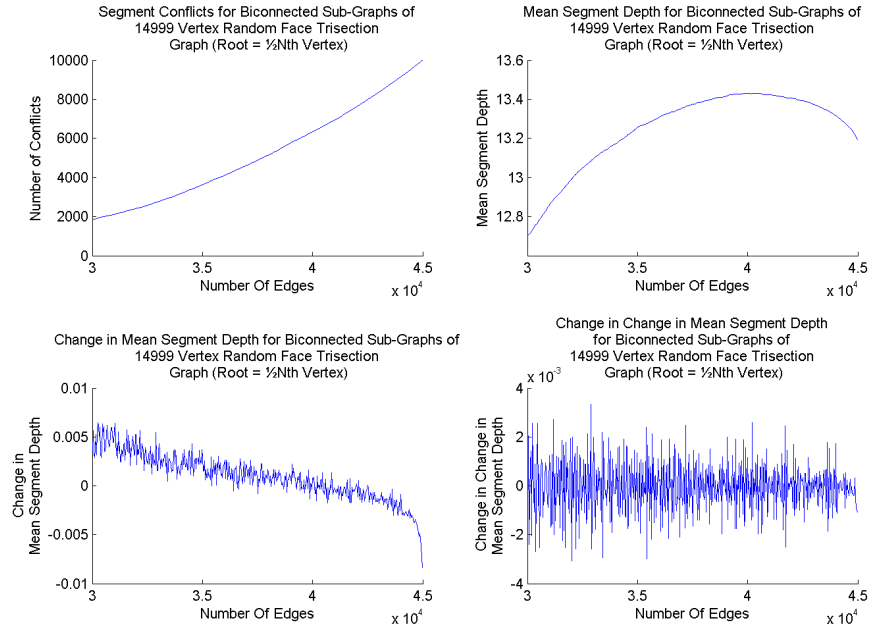


Figure A.4.17: Mean Segment Depth for Biconnected Sub-Graphs of Random Face Trisection Graph (14999 Vertices, ~ 30000 - 45000 Edges, $\frac{1}{2}N^{\text{th}}$ Vertex Root)

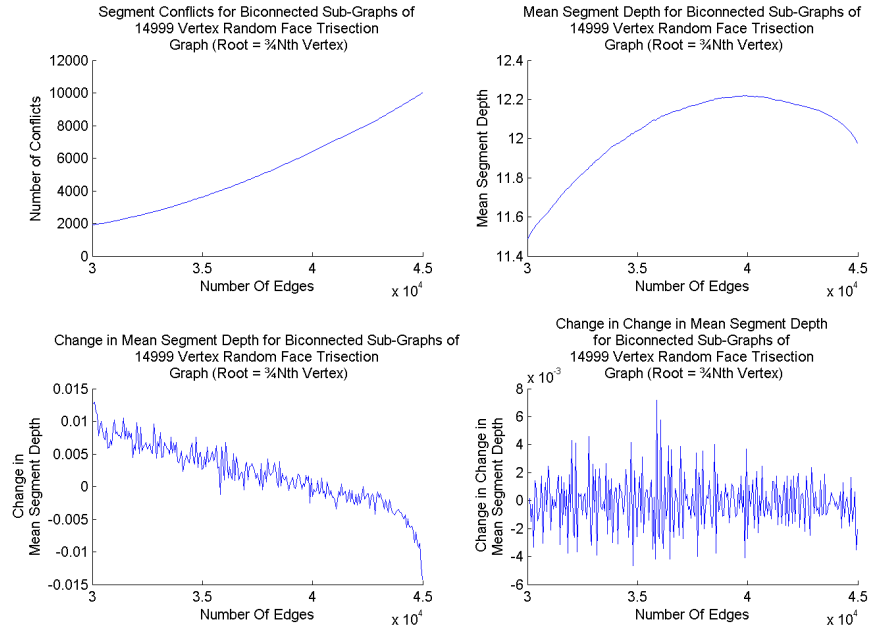


Figure A.4.18: Mean Segment Depth for Biconnected Sub-Graphs of Random Face Trisection Graph (14999 Vertices, ~ 30000 - 45000 Edges, $\frac{3}{4}N^{\text{th}}$ Vertex Root)

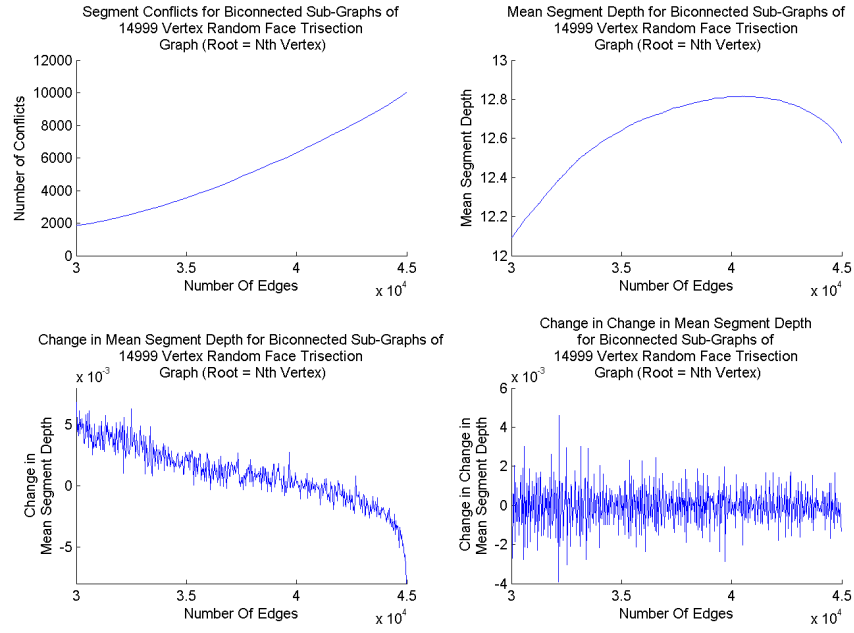


Figure A.4.19: Mean Segment Depth for Biconnected Sub-Graphs of Random Face Trisection Graph (14999 Vertices, ~ 30000 - 45000 Edges, N^{th} Vertex Root)

A.5 Variations in Size of Permutable, Flippable Graphs

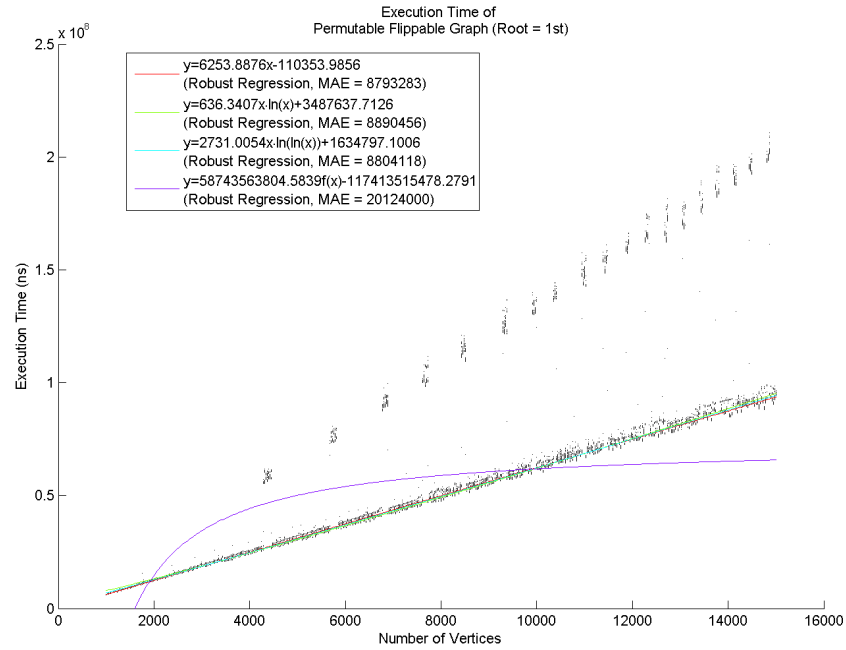
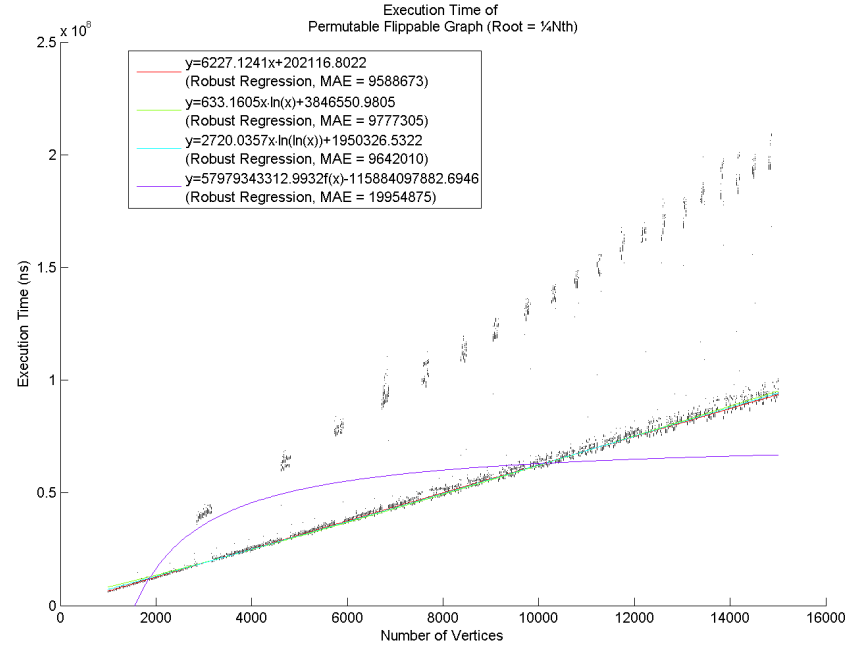
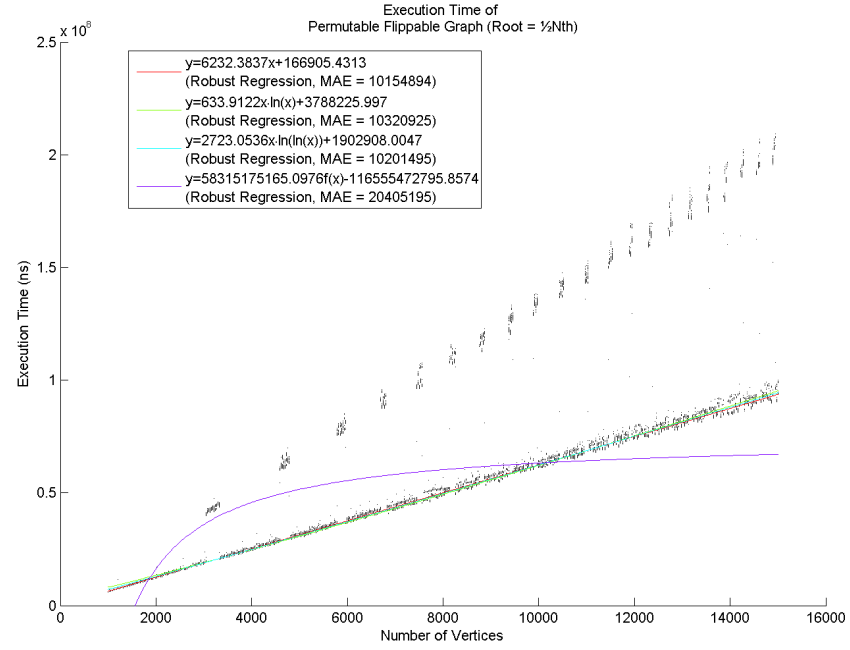
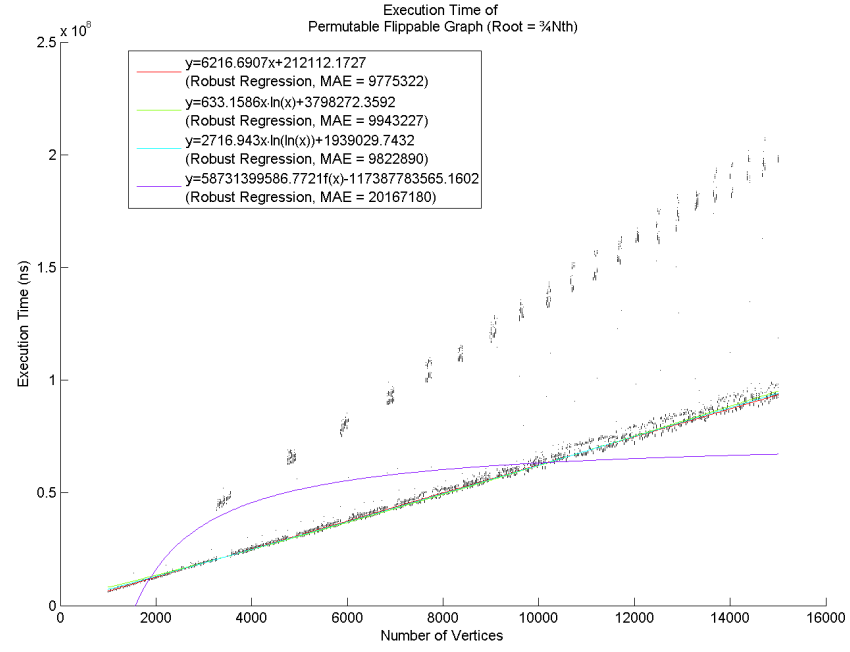
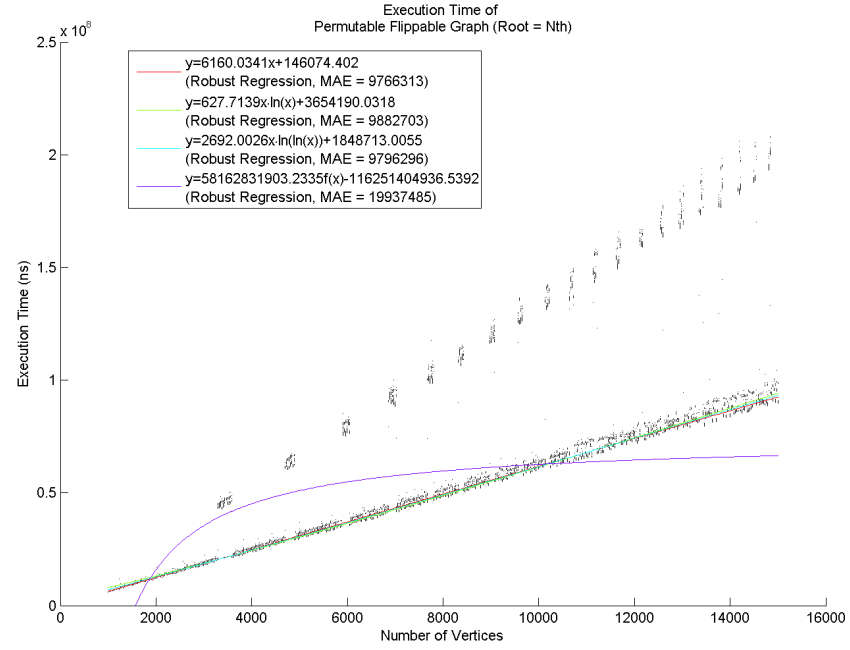


Figure A.5.1: Permutable, Flippable Graph (1002-15002 Vertices, 1^{st} Vertex Root)

Figure A.5.2: Permuable, Flippable Graph (1002-15002 Vertices, $\frac{1}{4}N^{\text{th}}$ Vertex Root)Figure A.5.3: Permuable, Flippable Graph (1002-15002 Vertices, $\frac{1}{2}N^{\text{th}}$ Vertex Root)

Figure A.5.4: Permuable, Flippable Graph (1002-15002 Vertices, $\frac{3}{4}N^{\text{th}}$ Vertex Root)Figure A.5.5: Permuable, Flippable Graph (1002-15002 Vertices, N^{th} Vertex Root)

Appendix B

Configuration used for Empirical Testing

The tests were performed on the following hardware and software configuration:

- Toshiba Satellite Pro Laptop A210;
- AMD Athlon 64 Dual-Core Processors, 1.8GHz;
- 2Gb RAM; and
- Windows Vista Business (6.0, Build 6002)

The software was written in Java and run from the Eclipse Platform; this choice was made as there is an inbuilt functionality to redirect the console output to a file and therefore the garbage collection output provided using the `PrintGCDetails` VM option will output inline with the normal program output. The following Eclipse configuration was used:

- Eclipse version: 3.4.2 (Build id: M20090211-1700);
- Java Virtual Machine: Java Development Kit 1.6.0_11;
- VM arguments: `-server -Xms1536M -Xmx1536M -XX:+PrintGCDetails`;
- Console allocated and redirected to file.

During execution the laptop was run in flight mode with all processes and services, not required for minimal operating system functionality, stopped to minimise external influences.

Appendix C

Clock Resolution Testing

C.1 Clock Resolution Test Code

```
1 import java.util.*;
2
3 public class ClockTest {
4     public static final int REPETITIONS = 10000000;
5
6     public static void main(String[] args) {
7         /*
8          * Fill the array with measurements between clock cycles.
9          */
10        final long[] data = new long[REPETITIONS];
11        for ( int i = 0; i < REPETITIONS; i++ ) {
12            long startTime = System.nanoTime();
13            long endTime = System.nanoTime();
14            data[i] = endTime - startTime;
15        }
16        /*
17         * Compile frequency data from the previous timing data.
18         */
19        HashMap<Long, Integer> frequencyData
20            = new HashMap<Long, Integer>();
21        for ( int i = 0; i < REPETITIONS; i++ ) {
22            if ( frequencyData.containsKey( data[i] ) )
23                frequencyData.put(
24                    data[i],
25                    frequencyData.get( data[i] ) + 1
26                );
27            else frequencyData.put( data[i], 1 );
28        }
29        Long[] times = frequencyData.keySet().toArray(
30            new Long[ frequencyData.keySet().size() ]
31        );
32        Arrays.sort( times );
33        boolean first = true;
34        /*
35         * Output the data in a MATLAB Friendly format.
36         */
37        System.out.print( '[' );
38        for ( long time: times ) {
39            if ( first ) first = false;
40            else System.out.print( "," );
41            System.out.print( " " + time +
```

```

42         " " + frequencyData.get( time ) );
43     }
44     System.out.println( ']' );
45 }
46 }

```

C.2 Clock Resolution Raw Data

Time (ns)	Frequency	Time (ns)	Frequency	Time (ns)	Frequency	Time (ns)	Frequency
977	555426	1466	2493535	1955	1813	2444	1277
978	1940982	1467	5000942	1956	2281	2445	1001

Time (ns)	F	Time (ns)	F	Time (ns)	F	Time (ns)	F	Time (ns)	F
2933	101	11733	8	19067	3	26888	1	34223	1
2934	48	11734	6	19555	2	26889	12	34711	6
3422	53	12222	10	19556	5	27377	1	35200	12
3423	17	12223	3	20044	5	27378	6	35688	1
3911	33	12711	9	20045	3	27866	4	35689	20
3912	3	12712	2	20533	7	27867	7	36177	6
4400	17	13200	8	20534	4	28355	3	36178	15
4889	4	13689	6	21022	7	28356	8	36666	4
6356	1	14177	1	21023	3	28844	7	36667	9
6844	7	14178	3	21511	11	28845	6	37155	2
6845	2	14666	1	21512	2	29333	13	37156	5
7333	26	14667	1	22000	14	29334	1	37644	4
7334	13	15155	2	22488	1	29822	8	37645	1
7822	22	15156	3	22489	21	29823	2	38133	5
7823	4	15644	3	22977	3	30311	7	38134	1
8311	16	15645	3	22978	11	30800	5	38622	1
8312	2	16133	8	23466	3	31289	8	38623	1
8800	24	16134	3	23467	7	31777	2	39111	8
9288	3	16622	1	23955	1	31778	2	39112	3
9289	35	16623	2	23956	13	32266	1	39600	7
9777	9	17111	2	24444	12	32267	2	40089	7
9778	39	17112	1	24445	7	32755	1	41066	2
10266	15	17600	10	24933	14	32756	2	41067	3
10267	28	18088	4	24934	3	33244	1	41555	1
10755	16	18089	10	25422	7	33245	3	41556	3
10756	16	18577	1	25423	5	33733	2	42044	1
11244	13	18578	13	25911	12	33734	2	42533	4
11245	12	19066	5	26400	12	34222	6	42534	1

Time (ns)	F	Time (ns)	F	Time (ns)	F	Time (ns)	F	Time (ns)	F
43022	4	58178	13	70888	2	82134	3	95333	1
43023	3	58666	5	70889	13	82622	3	95334	1
43511	2	58667	11	71377	3	82623	3	95822	3
44000	5	59155	8	71378	12	83111	2	96311	7
44489	3	59156	6	71866	5	83112	1	96800	7
44978	1	59644	13	71867	16	83600	4	97288	1
45466	1	59645	1	72355	9	84089	2	97289	7
45467	4	60133	10	72356	12	84577	1	97778	1
45955	1	60134	8	72844	4	84578	5	98267	2
45956	1	60622	12	72845	6	85066	3	98755	1
46444	2	60623	4	73333	8	85067	1	98756	2
46445	1	61111	16	73334	4	85555	2	99244	4
46933	2	61600	25	73822	16	85556	5	99245	2
46934	1	62089	18	73823	3	86044	3	99733	3
47422	3	62577	6	74311	9	86045	1	99734	1
47911	5	62578	16	74312	3	86533	4	100222	3
47912	2	63066	5	74800	5	86534	3	100711	6
48889	2	63067	18	75288	1	87022	4	101200	8
49377	1	63555	11	75289	10	87023	2	101688	1
49378	3	63556	4	75777	4	87511	5	101689	2
49867	2	64044	8	75778	6	88000	3	102177	1
50844	3	64045	4	76266	6	88488	1	102178	3
51333	1	64533	14	76267	9	88489	3	102666	2
51334	2	64534	4	76755	6	88977	2	102667	1
52311	2	65022	16	76756	5	88978	3	103155	2
52800	2	65023	7	77244	6	89466	1	103156	4
53289	1	65511	20	77245	5	89467	3	103644	1
53777	1	65512	4	77733	2	89955	2	103645	2
53778	3	66000	17	77734	2	89956	1	104133	1
54266	2	66489	21	78222	5	90444	1	104134	1
54267	8	66977	7	78223	1	90445	2	104622	5
54755	5	66978	21	78711	5	90933	5	104623	1
54756	4	67466	5	78712	2	90934	3	105111	1
55244	4	67467	13	79200	4	91422	2	105112	1
55245	4	67955	6	79688	1	91911	3	105600	3
55733	11	67956	13	79689	3	92400	3	106088	1
55734	2	68444	10	80177	1	92888	1	106089	2
56222	7	68445	10	80178	5	92889	5	106578	7
56223	5	68933	18	80666	1	93378	4	107066	1
56711	14	68934	11	80667	1	93866	5	107556	2
56712	5	69422	22	81155	1	93867	4	108044	4
57200	21	69423	8	81156	2	94355	1	108045	4
57688	1	69911	20	81644	4	94356	1	108533	2
57689	12	69912	2	81645	2	94844	3	109022	4
58177	2	70400	19	82133	4	94845	2	109511	3

Time (ns)	F	Time (ns)	F	Time (ns)	F	Time (ns)	F	Time (ns)	F
110000	2	128578	2	148622	2	176000	1	242489	1
110489	2	129067	2	149111	3	176489	2	247378	2
110978	3	129555	1	149112	1	176977	1	251778	1
111467	2	129556	1	149600	4	177956	2	254711	1
111955	3	130044	2	150089	2	178444	1	260088	1
111956	1	130045	1	150578	1	178934	2	265467	1
112444	2	130533	3	151066	1	179912	1	268400	1
112445	4	131022	1	151067	1	180400	2	272800	1
112933	1	131023	1	152045	1	181866	1	279645	1
113422	6	131511	2	152534	2	181867	2	282578	1
113911	3	132489	3	153022	1	182356	1	286489	2
114400	3	132978	1	153511	1	182845	2	287467	1
114888	1	133467	3	154000	1	183333	1	288933	1
114889	5	133956	2	154001	1	186756	1	293333	1
115378	1	134444	3	154489	1	187244	1	304578	1
115866	4	134445	2	155466	3	187245	1	306044	1
116356	3	135911	3	155467	3	188711	2	307511	1
116844	1	136400	2	155956	3	189689	1	310445	1
116845	1	136888	1	156444	2	190666	1	320711	1
117333	3	136889	1	156445	2	191156	2	326578	1
117334	2	137377	1	157422	1	191644	1	334400	1
118311	1	138356	2	157423	1	192133	1	337333	1
118312	1	138844	1	158400	3	192622	1	343689	1
118800	5	138845	1	159867	3	194089	1	352978	1
119288	1	139333	1	160355	1	198000	1	353956	1
119289	6	139334	1	160356	3	199467	1	358356	1
119777	1	139822	1	160844	2	201911	1	362755	1
119778	1	139823	1	160845	2	202889	1	366178	1
120267	3	140311	2	161333	1	203378	1	367644	1
120756	3	140312	2	161334	2	205822	1	371067	1
121244	1	140800	2	165244	1	207778	1	374000	1
121245	2	141289	3	165733	1	209244	1	378400	1
122222	3	141778	1	167200	1	211200	1	380356	1
122711	3	142266	1	167689	1	213156	1	384267	1
123200	1	143244	1	168177	1	213644	1	389155	1
124178	3	143245	1	168178	1	220489	2	396000	1
124666	2	143733	3	168666	1	224400	1	414577	1
124667	5	143734	2	168667	1	225378	1	416044	1
125155	1	144222	1	169155	1	230755	1	423377	1
125156	3	144711	2	170133	1	231245	1	429733	1
125644	3	145200	4	170623	1	231733	1	435111	1
125645	1	145689	1	171600	1	233200	1	435600	1
126622	3	146178	3	172577	1	235156	1	445866	1
127111	2	146667	3	173555	2	237111	1	446845	1
127600	1	147156	1	174533	1	238577	1	454177	1
128089	2	148133	1	174534	1	241511	1	454666	1

Time (ns)	F	Time (ns)	F	Time (ns)	F	Time (ns)	F	Time (ns)	F
464933	1	533378	1	555378	1	567600	1	677111	1
467378	1	535822	2	556845	1	574444	1	695689	1
474223	1	538756	2	559289	2	574933	1	696667	1
475200	1	540222	1	559778	1	593511	1	770489	1
477156	1	543155	1	560267	1	611600	1	798844	1
487423	1	543645	1	560756	1	619911	1	811556	1
492311	1	547556	1	563200	1	622356	1	821334	1
509422	1	549023	1	564178	1	623333	1	831112	1
512356	1	550978	1	565155	1	625778	1	6486579	1
522623	1	551466	1	565156	2	655111	1	17440136	1
530934	1	554889	1						

Appendix D

Java Source Code

D.1 Data Structure Files

D.1.1 mgtaylor.datastructures.AccessibleUnmodifiableLinkedList

```
1 package mgtaylor.datastructures;
2
3 import java.util.ConcurrentModificationException;
4
5 /**
6  * A linked list that makes the (normally private) data structures
7  * containing links between elements public and adds functionality
8  * to append elements to head/tailwards of a given list data
9  * structure.
10  * @author Martyn G. Taylor
11  * @param <E>
12  */
13 public class AccessibleUnmodifiableLinkedList<E>
14     extends UnmodifiableLinkedList<E> {
15     /**
16      *
17      * @return
18      */
19     public ListElement getHeadContainer() { return head; }
20
21     /**
22      *
23      * @return ListElement
24      */
25     public ListElement getTailContainer() { return tail; }
26
27     /**
28      * Overrides the superclass' findElement method to make it
29      * publicly accessible.
30      * @param element
31      * @return ListElement
32      */
33     @Override
34     public ListElement findElement( final E element ) {
35         return super.findElement( element );
36     }
37
38     /**
```

```

39      *
40      * @param listElement
41      * @param element
42      */
43      public void appendHeadwardsOf( ListElement tailwardsElement , E element ) {
44          if ( size() == 0 || tailwardsElement == head )
45              addHead( element );
46          else {
47              long change = changes;
48              ListElement le = new ListElement( element );
49              ListElement headwardsElement = tailwardsElement.getHeadwards();
50              le.setTailwards( tailwardsElement );
51              le.setHeadwards( headwardsElement );
52              tailwardsElement.setHeadwards( le );
53              headwardsElement.setTailwards( le );
54              size++;
55              if ( change != changes )
56                  throw new ConcurrentModificationException();
57              incrementChange();
58          }
59      }

61      /**
62      *
63      * @param headwardsElement
64      * @param element
65      */
66      public void appendTailwardsOf( ListElement headwardsElement , E element ) {
67          if ( size() == 0 || headwardsElement == tail )
68              addTail( element );
69          else {
70              long change = changes;
71              ListElement le = new ListElement( element );
72              ListElement tailwardsElement = headwardsElement.getTailwards();
73              le.setTailwards( tailwardsElement );
74              le.setHeadwards( headwardsElement );
75              tailwardsElement.setHeadwards( le );
76              headwardsElement.setTailwards( le );
77              size++;
78              if ( change != changes )
79                  throw new ConcurrentModificationException();
80              incrementChange();
81          }
82      }
83  }

```

D.1.2 mgtaylor.datastructures.LinkList

```

1 package mgtaylor.datastructures;

2
3 import java.util.Iterator;
4 import java.util.ConcurrentModificationException;

5
6 /**
7  * <p>An extension of the UnmodifiableLinkList structure that adds
8  * functionality to:</p>
9  * <ul>
10 * <li>Remove a given element from the list.</li>
11 * <li>Remove the head element from the list.</li>
12 * <li>Remove the tail element from the list.</li>
13 * <li>Clear (remove all elements from) the list.</li>
14 * <li>Remove the element at the current iterator position.</li>
15 * </ul>
16 * @author Martyn G. Taylor
17 * @see mgtaylor.datastructures.UnmodifiableLinkList UnmodifiableLinkList
18 * @param <E>
19 */
20 public class LinkList<E> extends UnmodifiableLinkList<E>
21     implements Iterable<E> {

22     /**
23      * Constructor to set List to be modifiable.
24      */
25     public LinkList() {
26         super();
27         this.isModifiable = true;
28     }

29
30     /**
31      * Removes a given list element from the list.
32      * @param le
33      * @return ListElementStore - A container used to store the
34      *         current and next elements of the list to keep a valid
35      *         state for an iterator.
36      */
37     protected ListElementStore removeElement( ListElement le ) {
38         if ( le == null )
39             return null;

40
41         if ( le == head && le == tail ) {
42             head = tail = null;
43             size = 0;
44             return new ListElementStore( null, null );
45         } else if ( le == head ) {
46             head = head.getTailwards();
47             if ( head != null )
48                 head.setHeadwards( null );
49             le.setTailwards( null );
50             size--;
51             return new ListElementStore( null, head );
52         } else if ( le == tail ) {
53             tail = tail.getHeadwards();
54             le.setHeadwards( null );
55             if ( tail != null )
56                 tail.setTailwards( null );
57             size--;
58             return new ListElementStore( tail, null );
59         } else {
60             final ListElement current = le.getHeadwards();
61             final ListElement next = le.getTailwards();

```



```

63         current.setTailwards( next );
64         next.setHeadwards( current );
65         le.setHeadwards( null );
66         le.setTailwards( null );
67         size--;
68         return new ListElementStore( current, next );
69     }
70 }

72 /**
73  * Removes all elements from the list.
74  * @throws ConcurrentModificationException Thrown if the list is
75  *     modified during the execution of this method.
76  * @throws IllegalArgumentException Thrown if the list cannot
77  *     be modified.
78  */
79 public void clear() {
80     if ( !isModifiable )
81         throw new IllegalArgumentException(
82             "mgtaylor.datastructures.LinkList.clear() - " +
83             "List is not modifiable." );

85     final long change = changes;

87     head = tail = null;
88     size = 0;
89     isModifiable = true;

91     if ( change != changes )
92         throw new ConcurrentModificationException(
93             "mgtaylor.datastructures.LinkList.clear()" );
94     incrementChange();
95 }

97 /**
98  * Removes a given element from the list.
99  * @param element
100  * @return boolean - Whether the element was successfully
101  *     found and removed.
102  * @throws ConcurrentModificationException Thrown if the list is
103  *     modified during the execution of this method.
104  * @throws IllegalArgumentException Thrown if the list cannot
105  *     be modified.
106  */
107 public boolean remove( final E element ) {
108     if ( element == null )
109         throw new IllegalArgumentException( "Null is an invalid argument." );
110     if ( !isModifiable )
111         throw new IllegalArgumentException( "List has been appended with an
            UnmodifiableLinkList and items cannot be removed from the list." );

113     final long change = changes;

115     final boolean found = removeElement( findElement( element ) ) != null;

117     if ( change != changes )
118         throw new ConcurrentModificationException();
119     incrementChange();

121     return found;
122 }

124 /**
125  * Removes the head element from the list.
126  * @return E - The removed element.

```

```

127  * @throws ConcurrentModificationException Thrown if the list is
128  *         modified during the execution of this method.
129  * @throws IllegalArgumentException Thrown if the list cannot
130  *         be modified.
131  */
132  public E removeHead() {
133      if ( !isModifiable )
134          throw new IllegalArgumentException( "List is not modifiable." );
135      final long change = changes;
136      final E element;
137      if ( size > 0 ) {
138          element = head.getElement();
139          if ( size == 1 )
140              head = tail = null;
141          else {
142              head = head.getTailwards();
143              head.setHeadwards( null );
144          }
145          size--;
146      } else
147          element = null;
148      if ( change != changes )
149          throw new ConcurrentModificationException();
150      incrementChange();
151      return element;
152  }

153  /**
154   * Removes the tail element from the list.
155   * @return E - The removed element.
156   * @throws ConcurrentModificationException Thrown if the list is
157   *         modified during the execution of this method.
158   * @throws IllegalArgumentException Thrown if the list cannot
159   *         be modified.
160   */
161  public E removeTail() {
162      if ( !isModifiable )
163          throw new IllegalArgumentException( "List is not modifiable." );
164      final long change = changes;
165      final E element;
166      if ( size > 0 ) {
167          element = tail.getElement();
168          if ( size == 1 )
169              head = tail = null;
170          else {
171              tail = tail.getHeadwards();
172              tail.setTailwards( null );
173          }
174          size--;
175      } else
176          element = null;
177      if ( change != changes )
178          throw new ConcurrentModificationException();
179      incrementChange();
180      return element;
181  }

182  /**
183   * <p>An iterator to loop over the list from head-to-tail.</p>
184   * <p>Overrides the iterator from the super-class to add
185   * functionality to remove elements from the list.</p>
186   * @return Iterator<E>
187   * @see mgtaylor.datastructures.UnmodifiableLinkedList#iterator()
188   *      UnmodifiableLinkedList.iterator()
189   */

```

```

192     @Override
193     public Iterator<E> iterator() {
194         return new Iterator<E>() {
195             ListElement current = null;
196             ListElement next    = head;

198             @Override
199             public boolean hasNext() {
200                 return next != null;
201             }

203             @Override
204             public E next() {
205                 if ( hasNext() ) {
206                     current = next;
207                     next = next.getTailwards();
208                     return current.getElement();
209                 } else
210                     return null;
211             }

213             /**
214              * @throws ConcurrentModificationException Thrown if the list
215              *         is modified during the execution of this method.
216              * @throws IllegalArgumentException Thrown if the list cannot
217              *         be modified; there are no elements to remove; or if
218              *         the iterator has not been moved from the initial
219              *         point.
220              */
221             @Override
222             public void remove() {
223                 if ( size() == 0 )
224                     throw new IllegalArgumentException( "List is empty; there is
225                     nothing to remove." );
226                 if ( current == null )
227                     throw new IllegalArgumentException( "Iterator is before the first
228                     list item; call next() before removing an item." );
229                 if ( !isModifiable )
230                     throw new IllegalArgumentException( "List has been appended with
231                     an UnmodifiableLinkedList and items cannot be removed from the
232                     list." );

233                 final long change = changes;

234                 ListElementStore elements = removeElement( current );
235                 if ( elements != null ) {
236                     current = elements.getCurrent();
237                     next = elements.getNext();
238                 } else {
239                     current = null;
240                     next = null;
241                 }

242                 if ( change != changes )
243                     throw new ConcurrentModificationException();
244                 incrementChange();
245             }
246         };
247     }

248     /**
249     * <p>An iterator to loop over the list from tail-to-head.</p>
250     * <p>Overrides the iterator from the super-class to add
251     * functionality to remove elements from the list.</p>
252     * @return Iterator<E>

```

```

253     * @see mgtaylor.datastructures.UnmodifiableLinkedList#iterator()
254     *     UnmodifiableLinkedList.iterator()
255     */
256     @Override
257     public Iterator<E> reverseIterator() {
258         return new Iterator<E>() {
259             ListElement current = null;
260             ListElement prev = tail;

261
262             @Override
263             public boolean hasNext() {
264                 return prev != null;
265             }

266
267             @Override
268             public E next() {
269                 if ( hasNext() ) {
270                     current = prev;
271                     prev = prev.getHeadwards();
272                     return current.getElement();
273                 } else
274                     return null;
275             }

276
277             /**
278              * @throws ConcurrentModificationException Thrown if the list
279              *         is modified during the execution of this method.
280              * @throws IllegalArgumentException Thrown if the list cannot
281              *         be modified; there are no elements to remove; or if
282              *         the iterator has not been moved from the initial
283              *         point.
284              */
285             @Override
286             public void remove() {
287                 if ( size() == 0 )
288                     throw new IllegalArgumentException( "List is empty; there is
289                                                         nothing to remove." );
289                 if ( current == null )
290                     throw new IllegalArgumentException( "Iterator is before the first
291                                                         list item; call next() before removing an item." );
291                 if ( !isModifiable )
292                     throw new IllegalArgumentException( "List is not modifiable." );

293
294                 final long change = changes;

295
296                 ListElementStore elements = removeElement( current );
297                 if ( elements != null ) {
298                     current = elements.getNext();
299                     prev = elements.getCurrent();
300                 } else {
301                     current = null;
302                     prev = null;
303                 }

304
305                 if ( change != changes )
306                     throw new ConcurrentModificationException();
307                 incrementChange();
308             }
309         };
310     }

311
312     /**
313      * An internal data structure used to store a valid state
314      * for current & next elements when removing elements from
315      * the list.

```

```
316     * @author Martyn G Taylor
317     */
318     protected class ListElementStore {
319         /**
320          * The current position in the list. When an element is
321          * removed then the previous position is used.
322          */
323         private final ListElement current;
324         /**
325          * The next element in the list.
326          */
327         private final ListElement next;
328
329         /**
330          * Initialiser for the store.
331          * @param current
332          * @param next
333          */
334         public ListElementStore( final ListElement current, final ListElement next
335         ) {
336             this.current = current;
337             this.next = next;
338         }
339
340         /**
341          * Gets the current element in the store.
342          * @return ListElement
343          */
344         public ListElement getCurrent() { return current; }
345         /**
346          * Gets the next element in the store.
347          * @return ListElement
348          */
349         public ListElement getNext() { return next; }
350     }
```

D.1.3 mgtaylor.datastructures.MutableUnmodifiableLinkedList

```

1 package mgtaylor.datastructures;

3 public class MutableUnmodifiableLinkedList<E> extends UnmodifiableLinkedList<E> {
4     /**
5      * The mutable list that will be re-ordered when cycling through
6      * permutations.
7      */
8     private UnmodifiableLinkedList<E> permutation = new
9         UnmodifiableLinkedList<E>();

10    /**
11     * The underlying list elements stored as an array.
12     * <ul>
13     * <li>The array is populated in the same order as the initial list; the
14     * K<sup>th</sup> element of the array maps to the K<sup>th</sup> element of the list.</li>
15     * <li>This allows for constant time access to and modification of the list
16     * structure
17     * using the array index.</li>
18     * </ul>
19     */
20    private ListElement[] elements = null;

21    /**
22     * An array storing the current position of each element in the elements
23     * array.
24     * <ul>
25     * <li>The index for this array is the initial order of the elements.</li>
26     * <li>Therefore the position of the element with the largest index will be
27     * stored
28     * in the last index of this array; conversely the position of the smallest
29     * indexed
30     * element will be in index 0 of the array.</li>
31     * </ul>
32     */
33    private int[] elementPositions = null;

34    /**
35     * An array storing the current element index at a given position in the
36     * current
37     * permutation.
38     * <ul>
39     * <li>The array is initialised with the position indexes increasing
40     * sequentially
41     * as the elements are initially ordered sequentially.</li>
42     * <li>After cycling through permutations, if the value of the array at index
43     * M is
44     * N then the M<sup>th</sup> element of the permutation list has been given
45     * an index
46     * of N; this allows the algorithm to find the element index of a
47     * neighbouring element.</li>
48     * </ul>
49     */
50    private int[] positionToElements = null;

51    /**
52     * The total number of permutations; for an input list of N items there are
53     * N! permutations.
54     */
55    private int totalPermutations = 0;

56    /**

```

```

50     * The index of the current permutation; ranging between 0 and the total
51     * number of permutations.
52     */
53     private int          currentPermutation    = 0;
54
55     // private int          minChanges          = Integer.MAX_VALUE;
56     // private int          maxChanges          = 0;
57     // private int          totalChanges        = 0;
58
59     /**
60     * Gets the total number of permutations.
61     * @return int The total number of permutations.
62     */
63     public int getTotalPermutations() {
64         if ( elementPositions == null && size() > 0 )
65             generatePermutationData();
66         return totalPermutations;
67     }
68
69     /**
70     * Returns the index of the current permutation.
71     * @return int The index of the current permutation.
72     */
73     public int getCurrentPermutation() { return currentPermutation; }
74
75     /**
76     * Gets the current permutation of the base list.
77     *
78     * @return E[] The current permutation.
79     */
80     public UnmodifiableLinkedList<E> getPermutation() {
81         if ( elementPositions == null && size() > 0 )
82             generatePermutationData();
83         return permutation;
84     }
85
86     /**
87     * Initialises the arrays of data used to iterate from one permutation to the
88     * next.
89     * <p>
90     * For a set with N elements and C constraints, this requires O(CN) time and
91     * O(N + C)
92     * memory.
93     */
94     @SuppressWarnings("unchecked")
95     private void generatePermutationData() {
96         elements          = new UnmodifiableLinkedList.ListElement[ size() ];
97         positionToElements = new int[ size() ];
98         elementPositions   = new int[ size() ];
99         totalPermutations = 1;
100
101         int i = 0;
102         for ( ListElement element = permutation.head; element != null; element =
103             element.getTailwards() ) {
104             elements[i]          = element;
105             positionToElements[i] = i;
106             elementPositions[i]   = i;
107             totalPermutations    *= ++i;
108         }
109
110         meetsConstraints = true;
111         for ( PermutationConstraint constraint: constraints )
112             meetsConstraints &= constraint.testConstraintAfterGeneration();
113     }

```

```

111  /**
112  * Gets the next permutation of the cycle.
113  * <p>
114  * <b>Steinhaus–Johnson–Trotter Permutation Algorithm:</b>
115  * <ul>
116  * <li>Elements are initially ordered with an ascending index number from
    left-to-right.</li>
117  * <li>Elements are given a direction of travel, initially every element is
    travelling
118  * leftwards.</li>
119  * <li>Elements are mobile if, in their direction of travel, the neighbouring
120  * element is not the end of the list or an element with a higher index
    number.</li>
121  * <li>The element with the highest index is tested first, if it is mobile
    then it is
122  * swapped with the neighbouring element in its current direction of
    travel;</li>
123  * <li>If the element is not mobile then elements with successively lower
    indices are
124  * tested until a mobile element is found; else the element with the lowest
    index is
125  * swapped.</li>
126  * <li>Once the element has been swapped then the direction of travel is
    reversed for all
127  * elements with a higher index.</li>
128  * </ul>
129  * This algorithm follows a predictable pattern:
130  * <ul>
131  * <li>For a set with N elements, when cycling through permutations, the
    largest indexed
132  * element will be moved on occasions when the current permutation index (C)
    is not
133  * exactly divisible by N ( $C \bmod N \neq 0$ ).</li>
134  * <li>This results in the pattern that:
135  * <ul>
136  * <li>The element with the largest index is moved from the right-to-left
    of the list,
137  * each of the  $(N - 1)$  swaps it takes to achieve this movement represents
    a possible
138  * permutation;</li>
139  * <li>On the  $N^{\text{th}}$  swap, when the element with the largest index
    is immobile
140  * at the left end of the list, the next highest indexed mobile element is
    exchanged
141  * with a neighbour and the direction of travel for the largest element is
    reversed;</li>
142  * <li>On the  $(N + 1)^{\text{th}}$  to  $(2N - 1)^{\text{th}}$  swaps, the
    largest element
143  * is moved back from left-to-right; and</li>
144  * <li>On the  $2N^{\text{th}}$  swap, when the element with the largest
    index is immobile
145  * at the right of the list, again the next highest indexed mobile element
    is moved.</li>
146  * <li>This process is repeated until all permutations have been cycled
    through.</li>
147  * </ul>
148  * <li>Therefore for  $(N - 1)$  out of N steps the element with the largest
    index will be
149  * moved and the direction of travel is moving left when  $(C \bmod 2N < N)$  or
    right
150  * otherwise.</li>
151  * <li>The element with the second largest index will move when  $(C \bmod N = 0)$ 
    and
152  *  $(C/N \bmod (N-1) \neq 0)$  and the direction of
    travel is left when

```



```

153 * (C/N mod 2(N - 1) &lt; (N - 1)) or right otherwise.</li>
154 * <li>In general, the K<sup>th</sup> indexed element of an N element set
    will move when
155 * C is exactly divisible by [ $\text{floor}(\frac{N}{M} = (k+1) \dots N$ ] and C is not
    exactly divisible by K;
156 * the direction of travel is left if (C.K!/N! mod 2K &lt; K) or right
    otherwise.</li>
157 * <li>The element with index of 1 will only move when no other elements are
    mobile; this
158 * occurs when (for a set of N elements) the (N! - 1)<sup>th</sup>
    permutation is reached
159 * and the next permutation in the sequence is the 0<sup>th</sup>
    permutation, thus
160 * resetting the permutation order back to match the original order. The
    1<sup>st</sup>
161 * element always moves left.</li>
162 * </ul>
163 *
164 * Speed of the algorithm (for a list with N elements):
165 * <ul>
166 * <li>There are N! permutations.</li>
167 * <li>[N! - (N - 1)!] of the permutations are reached by swapping the largest
168 * (N<sup>th</sup>) element. This requires a single iteration of the
    loop.</li>
169 * <li>Of the (N - 1)! remaining permutations, [(N - 1)! - (N - 2)!] are
    reached by
170 * swapping the second largest element. This requires an iteration of the
    loop to
171 * determine that the largest element is immobile and a second iteration to
    swap the
172 * element.</li>
173 * <li>The K<sup>th</sup> element is moved [K! - (K - 1)!] times during one
    entire cycle
174 * through all possible permutations. Moving the K<sup>th</sup> element
    requires (N - K + 1)
175 * iterations through the loop (checking that N<sup>th</sup> down to
    (K+1)<sup>th</sup>
176 * elements are immobile) and one final iteration to swap the K<sup>th</sup>
    element.</li>
177 * <li>The 1<sup>st</sup> element is moved once during the entire cycle, to
    go from the
178 * [N! - 1]<sup>th</sup> permutation back to the 0<sup>th</sup> original
    permutation. To
179 * move the 1<sup>st</sup> element, the loop is iterated over [N - 1] times;
    this check
180 * that the N<sup>th</sup> down to 2<sup>nd</sup> elements are immobile
    before the
181 * 1<sup>st</sup> element is swapped (once the previous elements are all
    immobile there
182 * is no requirement to check for mobility of the 1<sup>st</sup> element as
    there is only
183 * one permutation where this occurs and the element is always mobile in this
    case).</li>
184 * <li>For an N element set, to cycle through all possible (N!) permutations
    requires a
185 * total of [ $\text{floor}(\frac{N}{K} = 2 \dots N$ ] iterations through the loop and [N!]
    swaps.</li>
186 * </ul>
187 * <li>At N=3, there is a total of 6 permutations and the loop is iterated
    a total of 8 times giving an
188 * average cost of 1.33 iterations per permutation.</li>
189 * <li>At N=4, there are totals of 24 permutations and 32 iterations,
    again, giving an average cost of
190 * 1.33 iterations per permutation.</li>

```

```

191  *   <li>At N=5, there are totals of 120 permutations and 152 iterations ,
192  *   iterations per permutation.</li>
193  *   <li>As N tends towards infinity the average cost tends towards 1
194  *   iteration per permutation following the
195  *   approximation Average Cost =  $1 + \frac{1}{N}$ .</li>
196  * </ul></li>
197  * <li>The maximum cost for moving from one permutation to the next in the
198  * cycle is O(N).</li>
199  * </ul>
200  */
201  public void nextPermutation() {
202      if ( size() <= 1 )          return;
203      if ( elementPositions == null ) generatePermutationData();
204
205      int p = currentPermutation = (currentPermutation + 1) % totalPermutations;
206      for ( int n = size(); n > 1; n-- ) {
207          if ( p % n != 0 ) {
208              swapElement( n - 1, (p % (2 * n)) < n );
209              return;
210          }
211          p /= n;
212      }
213      swapElement( 0, true );
214  }
215
216  /**
217   * For a given permutation index (P) of a set of elements (<1, ... N>),
218   * the
219   * N<sup>th</sup> element will be in position (N - 1 - (P mod N)) if (P mod
220   * 2N < N)
221   * or in position (P mod N) otherwise. This leaves (N - 1) unassigned
222   * positions.
223   * <p>
224   * For the general K<sup>th</sup> element, that will be put in position
225   * (K - 1 - (P.K!/N! mod K)) if (P.K!/N! mod 2K < K) or in position (P.K!/N!
226   * mod K)
227   * otherwise. If any elements with a higher index than K have an equal or
228   * lower
229   * position then the position of the K<sup>th</sup> element is shifted right
230   * by one
231   * for each of those elements.
232   * <p>
233   * For example, Permutation 23 of the set <1, 2, 3, 4, 5>:
234   * <style>
235   * table { border-collapse: collapse; width: 100%; }
236   * table, tr, td, th { border: 1px solid black; }
237   * td, th { padding 2px; }
238   * </style>
239   * <table cellpadding="2">
240   * <colgroup>
241   * <col width="5%" />
242   * <col width="10%" />
243   * <col width="10%" />
244   * <col width="15%" />
245   * <col width="10%" />
246   * <col width="30%" />
247   * <col width="20%" />
248   * </colgroup>
249   * <tr>
250   * <th>K</th>
251   * <th>P.K!/N!</th>
252   * <th>P.K!/N! mod K</th>
253   * <th>P.K!/N! mod 2K < K</th>
254   * <th>Initial<br />Position</th>

```

```

247 * <th>Position<br />Shifts</th>
248 * <th>Permutation</th>
249 * </tr>
250 * <tr>
251 *     <td>5</td>
252 *     <td>23</td>
253 *     <td>3</td>
254 *     <td>True</td>
255 *     <td>4 - 3 = 1</td>
256 *     <td>&nbsp;</td>
257 *     <td>&lt;4, 5, null, null, null&gt;</td>
258 * </tr>
259 * <tr>
260 *     <td>4</td>
261 *     <td>23/5 = 4</td>
262 *     <td>0</td>
263 *     <td>False</td>
264 *     <td>0</td>
265 *     <td>&nbsp;</td>
266 *     <td>&lt;4, 5, null, null, null&gt;</td>
267 * </tr>
268 * <tr>
269 *     <td>3</td>
270 *     <td>23/20 = 1</td>
271 *     <td>1</td>
272 *     <td>True</td>
273 *     <td>2 - 1 = 1</td>
274 *     <td>1 âĖŠ 2 (Due to 4<sup>th</sup> Element @ Position 0)<br />2 âĖŠ 3 (5 @
275 *     <td>&lt;4, 5, null, 3, null&gt;</td>
276 * </tr>
277 * <tr>
278 *     <td>2</td>
279 *     <td>23/60 = 0</td>
280 *     <td>0</td>
281 *     <td>True</td>
282 *     <td>1 - 0 = 1</td>
283 *     <td>1 âĖŠ 2 (4 @ 0)<br />2 âĖŠ 3 (5 @ 1)<br>3 âĖŠ 4 (3 @ 3)</td>
284 *     <td>&lt;4, 5, null, 3, 2&gt;</td>
285 * </tr>
286 * <tr>
287 *     <td>1</td>
288 *     <td>23/120 = 0</td>
289 *     <td>0</td>
290 *     <td>True</td>
291 *     <td>0 - 0 = 0</td>
292 *     <td>0 âĖŠ 1 (4 @ 0)<br />1 âĖŠ 2 (5 @ 1)</td>
293 *     <td>&lt;4, 5, 1, 3, 2&gt;</td>
294 * </tr>
295 * </table>
296 * <p>
297 * The minimum shifts occurs when generating the 0<sup>th</sup> permutation:
298 * <table cellpadding="2">
299 * <colgroup>
300 * <col width="5%" />
301 * <col width="10%" />
302 * <col width="10%" />
303 * <col width="15%" />
304 * <col width="10%" />
305 * <col width="30%" />
306 * <col width="20%" />
307 * </colgroup>
308 * <tr>
309 *     <th>K</th>

```

```

310      * <th>P.K!/N!</th>
311      * <th>P.K!/N! mod K</th>
312      * <th>P.K!/N! mod 2K &lt; K</th>
313      * <th>Initial<br />Position</th>
314      * <th>Position<br />Shifts</th>
315      * <th>Permutation</th>
316      * </tr>
317      * <tr>
318      *      <td>5</td>
319      * <td>0</td>
320      * <td>0</td>
321      * <td>True</td>
322      * <td>4 - 0 = 4</td>
323      * <td>&nbsp;</td>
324      * <td>&lt;null, null, null, null, 5&gt;</td>
325      * </tr>
326      * <tr>
327      *      <td>4</td>
328      * <td>0/5 = 0</td>
329      * <td>0</td>
330      * <td>True</td>
331      * <td>3 - 0 = 3</td>
332      * <td>&nbsp;</td>
333      * <td>&lt;null, null, null, 4, 5&gt;</td>
334      * </tr>
335      * <tr>
336      *      <td>3</td>
337      * <td>0/20 = 0</td>
338      * <td>0</td>
339      * <td>True</td>
340      * <td>2 - 0 = 0</td>
341      * <td>&nbsp;</td>
342      * <td>&lt;null, null, 3, 4, 5&gt;</td>
343      * </tr>
344      * <tr>
345      *      <td>2</td>
346      * <td>0/60 = 0</td>
347      * <td>0</td>
348      * <td>True</td>
349      * <td>1 - 0 = 1</td>
350      * <td>&nbsp;</td>
351      * <td>&lt;null, 2, 3, 4, 5&gt;</td>
352      * </tr>
353      * <tr>
354      *      <td>1</td>
355      * <td>0/120 = 0</td>
356      * <td>0</td>
357      * <td>True</td>
358      * <td>0 - 0 = 0</td>
359      * <td>&nbsp;</td>
360      * <td>&lt;1, 2, 3, 4, 5&gt;</td>
361      * </tr>
362      * </table>
363      * <p>
364      * The maximum shifts occurs when the permutation is generated such that the
365      * elements are in reverse order:
366      * <table cellpadding="2">
367      * <colgroup>
368      * <col width="5%" />
369      * <col width="10%" />
370      * <col width="10%" />
371      * <col width="15%" />
372      * <col width="10%" />
373      * <col width="30%" />
374      * <col width="20%" />

```

```

374 * </colgroup>
375 * <tr>
376 *     <th>K</th>
377 * <th>P.K!/N!</th>
378 * <th>P.K!/N! mod K</th>
379 * <th>P.K!/N! mod 2K &lt; K</th>
380 * <th>Initial<br />Position</th>
381 * <th>Position<br />Shifts</th>
382 * <th>Permutation</th>
383 * </tr>
384 * <tr>
385 *     <td>5</td>
386 * <td>64</td>
387 * <td>4</td>
388 * <td>True</td>
389 * <td>4 - 4 = 0</td>
390 * <td>&nbsp;</td>
391 * <td>&lt;5, null, null, null&gt;</td>
392 * </tr>
393 * <tr>
394 *     <td>4</td>
395 * <td>64/5 = 12</td>
396 * <td>0</td>
397 * <td>False</td>
398 * <td>0</td>
399 * <td>0 âĖŠ 1 (5 @ 0)</td>
400 * <td>&lt;5, 4, null, null, null&gt;</td>
401 * </tr>
402 * <tr>
403 *     <td>3</td>
404 * <td>64/20 = 3</td>
405 * <td>0</td>
406 * <td>False</td>
407 * <td>0</td>
408 * <td>0 âĖŠ 1 (5 @ 0)<br />1 âĖŠ 2 (4 @ 1)</td>
409 * <td>&lt;5, 4, 3, null, null&gt;</td>
410 * </tr>
411 * <tr>
412 *     <td>2</td>
413 * <td>64/60 = 1</td>
414 * <td>1</td>
415 * <td>True</td>
416 * <td>1 - 1 = 0</td>
417 * <td>0 âĖŠ 1 (5 @ 0)<br />1 âĖŠ 2 (4 @ 1)<br />2 âĖŠ 3 (3 @ 2)</td>
418 * <td>&lt;5, 4, 3, 2, null&gt;</td>
419 * </tr>
420 * <tr>
421 *     <td>1</td>
422 * <td>64/120 = 0</td>
423 * <td>0</td>
424 * <td>True</td>
425 * <td>0 - 0 = 0</td>
426 * <td>0 âĖŠ 1 (5 @ 0)<br />1 âĖŠ 2 (4 @ 1)<br />2 âĖŠ 3 (3 @ 2)<br />3 âĖŠ 4
    (2 @ 3)</td>
427 * <td>&lt;5, 4, 3, 2, 1&gt;</td>
428 * </tr>
429 * </table>
430 * <ul>
431 * <li>The maximum number of position assignments plus shifts is  $[\hat{A}^{1/2}N^{\hat{A}} + \hat{A}^{1/2}N]$ .</li>
432 * <li>The minimum number of position assignments plus shifts is  $N$ .</li>
433 * <li>The average number of assignments per permutation, over a complete
    cycle
434 * through  $N!$  permutations, is  $[\hat{A}^{1/4}N^{\hat{A}} + \hat{A}^{3/4}N]$ .</li>

```

```

435     * </ul>
436     *
437     * @param p    The index of the permutation to jump to.
438     */
439     public void jumpToPermutation( final int p ) {
440         if ( size() <= 1 )           return;
441         if ( elementPositions == null ) generatePermutationData();

442
443         currentPermutation = p % totalPermutations;
444         if ( currentPermutation < 0 )
445             currentPermutation += totalPermutations;

446
447         int q = currentPermutation;
448         final AccessibleUnmodifiableLinkedList<Integer> order = new
         AccessibleUnmodifiableLinkedList<Integer>();
449         // int k = 0;
450         ListElement listElement = tail;
451         for ( int n = size(); n >= 1; n-- ) {
452             int position = ( q % ( 2 * n ) < n ) ? ( n - ( q % n ) - 1 ) : ( q % n );
453             // k++;
454             UnmodifiableLinkedList<Integer>.ListElement phe =
             order.getHeadContainer();
455             while ( phe != null && phe.getElement() <= position ) {
456                 phe = phe.getTailwards();
457                 position++;
458             // k++;
459             }
460             if ( phe == null )
461                 order.addTail( position );
462             else
463                 order.appendHeadwardsOf( phe, position );
464             positionToElements[ position ] = n - 1;
465             elementPositions[ n - 1 ] = position;
466             elements[ position ].setElement( listElement.getElement() );
467             listElement = listElement.getHeadwards();
468             q /= n;
469         }

470
471         meetsConstraints = true;
472         for ( PermutationConstraint constraint: constraints )
473             meetsConstraints &= constraint.testConstraintAfterGeneration();
474         // minChanges = Math.min( k, minChanges );
475         // maxChanges = Math.max( k, maxChanges );
476         // totalChanges += k;
477     }

478
479     /**
480     * Gets the previous permutation of the cycle.
481     * @see #nextPermutation()
482     */
483     public void previousPermutation() {
484         if ( size() <= 1 )           return;
485         if ( elementPositions == null ) generatePermutationData();

486
487         int p = currentPermutation;
488         currentPermutation = currentPermutation == 0 ? totalPermutations -
         1 : currentPermutation - 1;
489         for ( int n = size(); n > 1; n-- ) {
490             if ( p % n != 0 ) {
491                 swapElement( n - 1, ( p % ( 2 * n ) ) >= n );
492                 return;
493             }
494             p /= n;
495         }
496         swapElement( 0, false );

```

```

497     }

499     /**
500     * Swaps the position of the element at the given index and the element
501     * neighbouring it in the specified direction.
502     * <p>
503     * Constraint checking is O(C) where C is the number of constraints.
504     *
505     * @param mobileElement The index of the mobile element that is being swapped.
506     * @param direction      Whether the element at the given index is moving left
507     *                        or right (false).
508     */
509     protected void swapElement( final int mobileElement, final boolean direction
510     ) {
511         final int mobileElementPosition = elementPositions[ mobileElement ];
512         final int swapElementPosition = mobileElementPosition + (direction?-1:1);
513         final int swapElement          = positionToElements[ swapElementPosition ];

514         elementPositions[ mobileElement ]      = swapElementPosition;
515         elementPositions[ swapElement ]        = mobileElementPosition;
516         positionToElements[ mobileElementPosition ] = swapElement;
517         positionToElements[ swapElementPosition ] = mobileElement;

519         final E temp = elements[ mobileElementPosition ].getElement();
520         elements[ mobileElementPosition ].setElement( elements[
521         swapElementPosition ].getElement() );
522         elements[ swapElementPosition ].setElement( temp );

523         meetsConstraints = true;
524         final E before;
525         final E after;
526         if ( direction ) {
527             before = elements[ swapElementPosition ].getElement();
528             after = elements[ mobileElementPosition ].getElement();
529         } else {
530             before = elements[ mobileElementPosition ].getElement();
531             after = elements[ swapElementPosition ].getElement();
532         }
533         for ( final PermutationConstraint constraint: constraints )
534             meetsConstraints &= constraint.testConstraintAfterSwap( before, after );
535         handleSwap( before, after );
536     }

538     /**
539     * Method stub to allow sub-classes to handle swaps.
540     * @param earlier The element swapped to an earlier position in the list.
541     * @param later   The element swapped to an later position in the list.
542     */
543     protected void handleSwap( final E earlier, final E later ) {}

545     @Override
546     public void addTail( E element ) {
547         if ( elements != null ) throw new IllegalArgumentException( "Permutation
548         has been generated and elements cannot be added to the list." );
549         super.addTail( element );
550         permutation.addTail( element );
551     }

552     @Override
553     public void addHead( E element ) {
554         if ( elements != null ) throw new IllegalArgumentException( "Permutation
555         has been generated and elements cannot be added to the list." );
556         super.addHead( element );
557         permutation.addHead( element );

```

```

557     }

559     @Override
560     public void addAllTail( UnmodifiableLinkedList<E> list ) {
561         if ( elements != null ) throw new IllegalArgumentException( "Permutation
562             has been generated and elements cannot be added to the list." );
563         super.addAllTail( list );
564         permutation.addAllTail( list );
565     }

566     @Override
567     public void addAllHead( UnmodifiableLinkedList<E> list ) {
568         if ( elements != null ) throw new IllegalArgumentException( "Permutation
569             has been generated and elements cannot be added to the list." );
570         super.addAllHead( list );
571         permutation.addAllHead( list );
572     }

573     /*****
574     * Constraints
575     *****/

577     /**
578     * Flag to check whether the permutation meets the constraints.
579     */
580     protected boolean meetsConstraints = true;

582     /**
583     * The constraints that the permutation must meet to be valid.
584     */
585     protected final UnmodifiableLinkedList<PermutationConstraint> constraints = new
586         UnmodifiableLinkedList<PermutationConstraint>();

587     /**
588     * Adds a constraint to the permutation.
589     *
590     * @param before
591     * @param after
592     */
593     public void addConstraint( final E before, final E after ) {
594         constraints.addTail( new PermutationConstraint( before, after ) );
595     }

597     /**
598     * Checks whether the permutation meets all the constraints.
599     * @return boolean Whether the permutation meets all constraints.
600     */
601     public boolean isValidPermutation() { return meetsConstraints; }

603     /**
604     * Class storing a constraint on the permutation restricting the permutation
605     * to be
606     * valid only when a given element appears in the permutation before another
607     * given
608     * element.
609     * @author Martyn Taylor
610     */
611     protected class PermutationConstraint {
612         private final E elementBefore;
613         private final E elementAfter;
614         private boolean constraintMatched = true;

615         /**
616         * Adds a constraint to the permutation that one element should always be
617         * earlier in the order than another element.

```



```

617     * @param before The element that should be earlier in the permutation.
618     * @param after  The element that should be later in the permutation.
619     */
620     private PermutationConstraint( final E before, final E after ) {
621         if ( before == null ) throw new IllegalArgumentException(
622             "Constrained element cannot be null." );
623         if ( after == null ) throw new IllegalArgumentException( "Constrained
        element cannot be null." );
624         if ( before == after ) throw new IllegalArgumentException(
625             "Constrained elements cannot be equal." );
626         elementBefore = before;
627         elementAfter  = after;
628     }
629
630     /**
631     * Checks the elements being swapped to see if they match the constrained
632     * elements and if so, whether they are in the correct order.
633     * <ul>
634     * <li>If the swapped elements match the constrained elements and they are
635     * in
636     * the correct order after the swap then the constraint has been met;</li>
637     * <li>If the swapped elements match the constrained elements and they are
638     * in
639     * the reverse order then the constraint is not matched and the
640     * permutation is
641     * invalid; and</li>
642     * <li>If one or more elements does not match the constrained elements then
643     * the swap will not change the relative order of the constrained elements
644     * and the validity of the permutation will be the same as the previous
645     * permutation with regards to this constraint.</li>
646     * </ul>
647     *
648     * @param swapFirst The swapped element that is now earlier in the
649     * permutation.
650     * @param swapSecond The swapped element that is now later in the
651     * permutation.
652     * @return boolean Whether the constraint has been matched.
653     */
654     private boolean testConstraintAfterSwap( final E swapFirst, final E
        swapSecond ) {
655         if ( swapFirst == elementBefore && swapSecond == elementAfter )
656             constraintMatched = true;
657         else if ( swapFirst == elementAfter && swapSecond == elementBefore )
658             constraintMatched = false;
659         return constraintMatched;
660     }
661
662     /**
663     * Iterates through the list of elements to find which if the constrained
664     * elements is first in the list. If the element constrained to be earlier
665     * in
666     * the permutation is found first then the constraint is matched;
667     * otherwise, the
668     * permutation is invalid.
669     *
670     * @return boolean Whether the constraint has been matched.
671     */
672     private boolean testConstraintAfterGeneration() {
673         for ( final E element:
674             PermutableUnmodifiableLinkedList.this.getPermutation() ) {
675             if ( element == elementBefore ) {
676                 constraintMatched = true;
677                 return true;
678             } else if ( element == elementAfter ) {
679                 constraintMatched = false;

```

```

670         return false;
671     }
672 }
673     throw new IllegalArgumentException( "No constraint elements found." );
674 }
675 }

677 /*
678     public static void main( String[] args ) {
679         for ( int n = 5; n <= 5; n++ ) {
680             PermutableUnmodifiableLinkedList<Integer> p = new
681             PermutableUnmodifiableLinkedList<Integer>();
682             PermutableUnmodifiableLinkedList<Integer> q = new
683             PermutableUnmodifiableLinkedList<Integer>();
684             Integer[] pelements = new Integer[ n ];
685             Integer[] qelements = new Integer[ n ];
686             for ( int m = 1; m <= n; m++ ) {
687                 p.addTail( pelements[ m - 1 ] = new Integer( m ) );
688                 q.addTail( qelements[ m - 1 ] = new Integer( m ) );
689             }
690             p.addConstraint( pelements[ 0 ], pelements[ 1 ] );
691             p.addConstraint( pelements[ 1 ], pelements[ 2 ] );
692             p.addConstraint( pelements[ 3 ], pelements[ 4 ] );
693
694             q.addConstraint( qelements[ 0 ], qelements[ 1 ] );
695             q.addConstraint( qelements[ 1 ], qelements[ 2 ] );
696             q.addConstraint( qelements[ 3 ], qelements[ 4 ] );
697
698             //      System.out.println( p.getCurrentPermutation() + ":\t" +
699             p.getPermutation() + '\t' + p.getCurrentPermutation() + ":\t" +
700             q.getPermutation() );
701             for ( int i = 1; i <= p.getTotalPermutations(); i++ ) {
702                 p.nextPermutation();
703                 q.jumpToPermutation( i );
704                 String ps = p.getPermutation().toString();
705                 String qs = q.getPermutation().toString();
706                 if ( ps.compareTo( qs ) != 0 || p.isValidPermutation() !=
707                 q.isValidPermutation() )
708                     System.out.println( p.getCurrentPermutation() + ":\t" + ps + " ("
709                     + p.isValidPermutation() + ")\t" + q.getCurrentPermutation() +
710                     ":\t" + qs + " (" + q.isValidPermutation() + ")" );
711                 System.out.println( p.getCurrentPermutation() + "/" +
712                 p.getTotalPermutations() + ":\t" + p.getPermutation() + "\t" +
713                 p.isValidPermutation() );
714                 if ( qs.compareTo( "<10, 9, 8, 7, 6, 5, 4, 3, 2, 1>" ) == 0 )
715                     System.out.println( p.getCurrentPermutation() + "/" +
716                     p.getTotalPermutations() + ":\t" + p.getPermutation() + "\t" +
717                     p.isValidPermutation() );
718             }
719             //      System.out.println( "Min: " + q.minChanges + "\tMax: " + q.maxChanges +
720             "\tTotal: " + q.totalChanges + "\tPerms: " + q.totalPermutations + "\t" +
721             ((float) q.totalChanges/q.totalPermutations));
722         }
723     }
724 }

```

D.1.4 mgtaylor.datastructures.UnmodifiableLinkedList

```

1 package mgtaylor.datastructures;

3 import java.util.ConcurrentModificationException;
4 import java.util.Iterator;

6 /**
7  * <p>A doubly linked list structure that allows elements of the given
8  * enumerated type
9  * to be added to, or retrieved from, the head or tail of the list and can
10  * iterate over
11  * the entire list. The structure has no capability to: remove items from the
12  * list; edit
13  * the order of the list; insert items in the middle of the list; or directly
14  * access
15  * elements in the middle of the list.</p>
16  *
17  * <p>The structure can be accessed in Constant time to:</p>
18  * <ul>
19  * <li>Add an element to the head or tail;</li>
20  * <li>Concatenate this list with another of the same enumerated type;</li>
21  * <li>Retrieve the head or tail element of the list; and</li>
22  * <li>Get the next element in the list from the iterator.</li>
23  * </ul>
24  * <p>The structure can be accessed in Linear time to:</p>
25  * <ul>
26  * <li>Query the list to see if it contains an element; and</li>
27  * <li>Get a string representation of the list.</li>
28  * </ul>
29  * @author Martyn Taylor
30  *
31  * @param <E>
32  */
33 public class UnmodifiableLinkedList<E> implements Iterable<E> {
34     /**
35      * Whether elements can be removed from the list.
36      */
37     protected boolean isModifiable = false;

38     /**
39      * The wrapper for the head element of the list.
40      */
41     protected ListElement head = null;

42     /**
43      * The wrapper for the tail element of the list.
44      */
45     protected ListElement tail = null;

46     /**
47      * The number of elements in the list.
48      */
49     protected int size = 0;

50     /**
51      * A reference variable to track changes to the data structure and
52      * is used to highlight concurrent modification exceptions.
53      */
54     protected long changes = Long.MIN_VALUE;

55     /**
56      * Checks whether elements can be removed from the list.
57      * @return boolean
58      */

```

```

59     */
60     public boolean canRemoveElements() {
61         return isModifiable;
62     }

64     /**
65      * Checks whether there are no elements in the list.
66      * @return boolean
67      */
68     public boolean isEmpty() {
69         return size() == 0;
70     }

72     /**
73      * Add an element to the head of the list.
74      *
75      * @param element – The non-null element to be added.
76      * @throws ConcurrentModificationException Thrown if the list is modified
77      *         during the execution of this method.
78      * @throws IllegalArgumentException Thrown if the head of the list is
79      *         connected
80      *         to another list and cannot be appended to.
81      */
82     public void addHead( E element ) {
83         if ( element == null )
84             throw new IllegalArgumentException( "Cannot add a null element." );
85         if ( head != null && head.getHeadwards() != null )
86             throw new IllegalArgumentException( "Head element is connected to
87             another list and cannot be appended to." );
88         long change = changes;
89         if ( size == 0 ) {
90             head = tail = new ListElement( element );
91             le.setTailwards( head );
92             head.setHeadwards( le );
93             head = le;
94         }
95         size++;
96         if ( change != changes )
97             throw new ConcurrentModificationException();
98         incrementChange();
99     }

101     /**
102      * Add an element to the tail of the list.
103      *
104      * @param element – The non-null element to be added.
105      * @throws ConcurrentModificationException Thrown if the list is modified
106      *         during the execution of this method.
107      * @throws IllegalArgumentException Thrown if the tail of the list is
108      *         connected
109      *         to another list and cannot be appended to.
110      */
111     public void addTail( E element ) {
112         if ( element == null )
113             throw new IllegalArgumentException( "Cannot add a null element." );
114         if ( tail != null && tail.getTailwards() != null )
115             throw new IllegalArgumentException( "Tail element is connected to
116             another list and cannot be appended to." );
117         long change = changes;
118         if ( size == 0 ) {
119             head = tail = new ListElement( element );

```

```

120         le.setHeadwards( tail );
121         tail.setTailwards( le );
122         tail = le;
123     }
124     size++;
125     if ( change != changes )
126         throw new ConcurrentModificationException();
127     incrementChange();
128 }

130 /**
131  * Returns the head element from the list or null if the list is empty.
132  *
133  * @return <E>;
134  */
135 public E getHead() {
136     if ( size == 0 )
137         return null;
138     return head.getElement();
139 }

141 /**
142  * Returns the tail element from the list or null if the list is empty.
143  *
144  * @return <E>;
145  */
146 public E getTail() {
147     if ( size == 0 )
148         return null;
149     return tail.getElement();
150 }

152 /**
153  * Returns the element before the tail element from the list or null if
154  * the list does not have at least two elements.
155  *
156  * @return <E>;
157  */
158 public E getPenultimateTail() {
159     if ( size < 2 )
160         return null;
161     return tail.getHeadwards().getElement();
162 }

164 /**
165  * Returns the size of the list.
166  * @return int
167  */
168 public int size() {
169     return size;
170 }

172 /**
173  * Appends the contents of the given list to the head of this list.
174  *
175  * @param list - A non-null list of the same enumerated type.
176  * @throws IllegalArgumentException Thrown if the head of this list or the
177  *         tail
178  *         of the given list are connected to another list and, therefore,
179  *         cannot
180  *         be appended to.
181  */
182 public void addAllHead( UnmodifiableLinkedList<E> list ) {
183     if ( list == null )
184         throw new IllegalArgumentException( "Cannot append a null list." );

```

```

183         if ( head != null && head.getHeadwards() != null )
184             throw new IllegalArgumentException( "This list's head element is
connected to another list and cannot be appended to." );
185         if ( list.tail != null && list.tail.getTailwards() != null )
186             throw new IllegalArgumentException( "Given list's tail element is
connected to another list and cannot be appended to." );
187         if ( list.size() == 0 )
188             return;
189         long change = changes;
190         long listChange = list.changes;

192         isModifiable &= list.isModifiable;

194         if ( size == 0 ) {
195             head = list.head;
196             tail = list.tail;
197             size = list.size;
198         } else {
199             head.setHeadwards( list.tail );
200             list.tail.setTailwards( head );
201             head = list.head;
202             size += list.size;
203         }

205         list.head = list.tail = null;
206         list.size = 0;

208         if ( change != changes || listChange != list.changes )
209             throw new ConcurrentModificationException();
210         incrementChange();
211     }

213     /**
214     * Appends the contents of the given list to the tail of this list.
215     *
216     * @param list - A non-null list of the same enumerated type.
217     * @throws IllegalArgumentException Thrown if the tail of this list or the
head
218     *         of the given list are connected to another list and, therefore,
cannot
219     *         be appended to.
220     */
221     public void addAllTail( UnmodifiableLinkList<E> list ) {
222         if ( list == null )
223             throw new IllegalArgumentException( "Cannot append a null list." );
224         if ( tail != null && tail.getTailwards() != null )
225             throw new IllegalArgumentException( "This list's tail element is
connected to another list and cannot be appended to." );
226         if ( list.head != null && list.head.getHeadwards() != null )
227             throw new IllegalArgumentException( "Given list's head element is
connected to another list and cannot be appended to." );
228         if ( list.size() == 0 )
229             return;
230         long change = changes;
231         long listChange = list.changes;

233         isModifiable &= list.isModifiable;

235         if ( size == 0 ) {
236             head = list.head;
237             tail = list.tail;
238             size = list.size;
239         } else {
240             tail.setTailwards( list.head );
241             list.head.setHeadwards( tail );

```

```

242         tail = list.tail;
243         size += list.size;
244     }

246     list.head = list.tail = null;
247     list.size = 0;

249     if ( change != changes || listChange != list.changes )
250         throw new ConcurrentModificationException();
251     incrementChange();
252 }

254 /**
255  * Checks whether a given element is contained within the list.
256  * @param element - The element to locate in the list.
257  * @return boolean
258  */
259 public boolean contains( final E element ) {
260     return findElement( element ) != null;
261 }

263 /**
264  * Finds the internal list data structure containing the given element
265  * within the list.
266  * @param element - The element to locate in the list.
267  * @return ListElement
268  */
269 protected ListElement findElement( final E element ) {
270     ListElement current = head;
271     while ( current != null ) {
272         if ( current.getElement() == element )
273             return current;
274         current = current.getTailwards();
275     }
276     return null;
277 }

279 /**
280  * Returns a comma-space separated sequence representing the contents
281  * of the list. The return value is bracketed by angle brackets (<>).
282  * @see UnmodifiableLinkedList#toString(String) toString( String )
283  * @return String
284  */
285 public String toString() {
286     return toString( " , " );
287 }

289 /**
290  * Returns a sequence representing the contents of the list. The return
291  * value is bracketed by angle brackets (<>) and pairs of elements are
292  * separated by the value of the function's separator argument.
293  * @param separator - A string defining the separator between list
294  *                  elements.
295  * @return String
296  */
297 public String toString( String separator ) {
298     StringBuffer b = new StringBuffer();
299     b.append( '<' );
300     boolean first = true;
301     for ( E e: this ) {
302         if ( first )
303             first = false;
304         else
305             b.append( separator );
306         b.append( e );

```

```

307     }
308     b.append( '>' );
309     return b.toString();
311 }
313 /**
314  * <p>Increments the counter recording the number of changes to the
315  * data structure.</p>
316  * <p>Used as a simple (imperfect) method to detect concurrent
317  * modification of a list.</p>
318  */
319 protected synchronized void incrementChange() {
320     if ( changes == Long.MAX_VALUE )
321         changes = Long.MIN_VALUE;
322     else
323         changes++;
324 }
326 /**
327  * <p>The data structure used to store a single element of data within
328  * the list</p>
329  * @author Martyn G. Taylor
330  */
331 protected class ListElement {
332     private E element;
333     private ListElement tailwards = null;
334     private ListElement headwards = null;
336     protected ListElement( E element ) {
337         this.element = element;
338     }
340     protected void setElement( E element ) { this.element = element; }
341     protected void setTailwards( ListElement next ) { this.tailwards = next; }
342     protected void setHeadwards( ListElement prev ) { this.headwards = prev; }
344     protected E getElement() { return element; }
345     protected ListElement getTailwards() { return tailwards; }
346     protected ListElement getHeadwards() { return headwards; }
347 }
349 /**
350  * Returns an iterator to loop through the list from head-to-tail.
351  * @return Iterator<E>
352  */
353 @Override
354 public Iterator<E> iterator() {
355     return new Iterator<E>() {
356         ListElement current = head;
358         @Override
359         public boolean hasNext() {
360             return current != null;
361         }
363         @Override
364         public E next() {
365             if ( current == null )
366                 return null;
367             E next = current.getElement();
368             current = current.getTailwards();
369             return next;
370         }
371     }

```



```
372         @Override
373         public void remove() {
374             throw new IllegalArgumentException( "Cannot remove from this list."
375             );
376         }
377     }
378
379     /**
380     * Returns an iterator to loop through the list from tail-to-head.
381     * @return Iterator<E>
382     */
383     public Iterator<E> reverseIterator() {
384         return new Iterator<E>() {
385             ListElement current = tail;
386
387             @Override
388             public boolean hasNext() {
389                 return current != null;
390             }
391
392             @Override
393             public E next() {
394                 if ( current == null )
395                     return null;
396                 E next = current.getElement();
397                 current = current.getHeadwards();
398                 return next;
399             }
400
401             @Override
402             public void remove() {
403                 throw new IllegalArgumentException( "Cannot remove from this list."
404                 );
405             }
406         }
407     }
```

D.2 Graph Files

D.2.1 mgtaylor.graph.Block

```

1 package mgtaylor.graph;

3 import mgtaylor.datastructures.LinkList;
4 import mgtaylor.datastructures.UnmodifiableLinkList;

6 /**
7  *
8  * @author Martyn Taylor
9  *
10 */
11 public class Block {
12     public static final boolean INSIDE          = true;
13     public static final boolean OUTSIDE         = !INSIDE;

15     private LinkList<Segment> inside             = new LinkList<Segment>();
16     private LinkList<Segment> outside            = new LinkList<Segment>();
17     private UnmodifiableLinkList<Segment> embeddedInside = new
UnmodifiableLinkList<Segment>();
18     private UnmodifiableLinkList<Segment> embeddedOutside = new
UnmodifiableLinkList<Segment>();

20     private final Segment parent;
21     private final Block previous;
22     private boolean flippable;

24     private boolean inverted          = false;

26     /**
27      * Creates a new block with the given segment on the inside.
28      * @param segment The non-null segment starting the block.
29      */
30     public Block( final Segment segment ) {
31         if ( segment == null ) throw new IllegalArgumentException( "Segment
cannot be null." );
32         parent                = segment.getParent();
33         previous = parent.getLastChildBlock();
34         flippable            = segment.isFlippable() && (parent.getParent() != null ||
parent.getChildSegments().getHead() != segment);
35         inside.addTail( segment );
36         embeddedInside.addTail( segment );

38         segment.getGraph().addBlock( this );
39         parent.addChildBlock( this );
40         parent.setLastEmbeddedSegment( segment );
41         if ( Graph.DEBUG_EMBED ) System.out.println( "B: New Block " +
segment.toString() );
42     }

44     /**
45      * Gets the parent segment for the block.
46      * @return Segment The non-null parent for the block.
47      */
48     public Segment getParent()          { return parent; }

50     /**
51      * Gets the previous block with the same parent.
52      * @return Block The previous block or null if there was no block that
53      *             was not already fully embedded when this block was added.
54      */

```

```

55     public Block getPrevious()                { return previous; }

57     /**
58      * Checks whether the block is flippable. If all the segments in the
59      * block are flippable then the block is flippable.
60      * @return boolean Whether the block is flippable.
61      */
62     public boolean isFlippable()                { return flippable; }

64     /**
65      * Gets the list of segments on the inside of the block that are not
66      * currently fully processed.
67      * @return {@link LinkedList}&lt;{@link Segment}&gt;;
68      */
69     public LinkedList<Segment> getInsideSegments() { return inside; }

71     /**
72      * Gets the list of segments on the outside of the block that are not
73      * currently fully processed.
74      * @return {@link LinkedList}&lt;{@link Segment}&gt;;
75      */
76     public LinkedList<Segment> getOutsideSegments() { return outside; }

78     /**
79      * Checks whether the inside of the block is empty.
80      * @return boolean True if it is empty, false otherwise.
81      */
82     public boolean isEmptyInside()                { return inside.isEmpty(); }

84     /**
85      * Checks whether the outside of the block is empty.
86      * @return boolean True if it is empty, false otherwise.
87      */
88     public boolean isEmptyOutside()                { return outside.isEmpty(); }

90     /**
91      * Checks whether there are no segments (either embedded or fully embedded)
92      * on the inside of the block.
93      * @return boolean True if it is empty, false otherwise.
94      */
95     public boolean hasNoSegmentsInside()                { return embeddedInside.isEmpty(); }

97     /**
98      * Checks whether there are no segments (either embedded or fully embedded)
99      * on the outside of the block.
100     * @return boolean True if it is empty, false otherwise.
101     */
102     public boolean hasNoSegmentsOutside()                { return embeddedOutside.isEmpty(); }

104     /**
105      * Gets the last segment embedded (but not fully embedded) on the
106      * inside of the block.
107      * @return Segment The latest segment embedded on the inside of the
108      *         block or null if the inside is empty.
109      */
110     public Segment getLastInside()                { return inside.getTail(); }

112     /**
113      * Gets the last segment embedded (but not fully embedded) on the
114      * outside of the block.
115      * @return Segment The latest segment embedded on the outside of the
116      *         block or null if the outside is empty.
117      */
118     public Segment getLastOutside()                { return outside.getTail(); }

```

```

120  /**
121   * Adds the given segment to the list of segments on the inside of the block.
122   * @param segment A non-null segment that shares the same parent as the block.
123   */
124  public void addSegmentInside(Segment segment) {
125      if ( segment == null )                throw new IllegalArgumentException(
126          "Segment cannot be null." );
127      if ( segment.getParent() != getParent() ) throw new
128          IllegalArgumentException( "Block and segment have different parents." );
129      if ( !segment.isFlippable() )
130          flippable = false;
131      inside.addTail( segment );
132      embeddedInside.addTail( segment );
133      parent.setLastEmbeddedSegment( segment );
134      if ( Graph.DEBUG_EMBED ) System.out.println( "B: Inside " +
135          segment.toString() );
136  }
137
138  /**
139   * Adds the given segment to the list of segments on the inside of the block.
140   * @param segment A non-null segment that shares the same parent as the block.
141   */
142  public void addSegmentOutside(Segment segment) {
143      if ( segment == null )                throw new IllegalArgumentException(
144          "Segment cannot be null." );
145      if ( segment.getParent() != getParent() ) throw new
146          IllegalArgumentException( "Block and segment have different parents." );
147      outside.addTail(segment);
148      embeddedOutside.addTail(segment);
149      parent.setLastEmbeddedSegment( segment );
150      if ( Graph.DEBUG_EMBED ) System.out.println( "B: Outside " +
151          segment.toString() );
152  }
153
154  /**
155   * Checks if the last (non-fully embedded) segment on the inside
156   * conflicts with the given segment.
157   * @param segment The non-null segment to check for a conflict.
158   * @return boolean True is the segment conflicts, false otherwise.
159   */
160  public boolean insideConflictsWith(Segment segment) {
161      if ( segment == null ) throw new IllegalArgumentException( "Segment
162          cannot be null." );
163      return !inside.isEmpty() && inside.getTail().conflictsWith(segment);
164  }
165
166  /**
167   * Checks if the last (non-fully embedded) segment on the outside
168   * conflicts with the given segment.
169   * @param segment The non-null segment to check for a conflict.
170   * @return boolean True is the segment conflicts, false otherwise.
171   */
172  public boolean outsideConflictsWith(Segment segment) {
173      if ( segment == null ) throw new IllegalArgumentException( "Segment
174          cannot be null." );
175      return !outside.isEmpty() && outside.getTail().conflictsWith(segment);
176  }
177
178  /**
179   * Finds which side of the block has the given segment and appends the given
180   * list of segments
181   * to the appropriate list. The segments are not added to the list of fully
182   * embedded segments

```

```

173     * as they are already embedded in another block which contains this
174     information.
175     * @param segment A non-null segment matching the last segment on one side of
176     the block.
177     * @param segments A non-null list of segments to be appended after the
178     matched segment.
179     */
180     public void appendSegmentsAfter( Segment segment, LinkedList<Segment> segments
181     ) {
182         if ( segment == null ) throw new IllegalArgumentException( "Segment
183         argument cannot be null." );
184         if ( segments == null ) throw new IllegalArgumentException( "Segment list
185         argument cannot be null." );
186         if ( inside.getTail() == segment ) {
187             inside.addAllTail( segments );
188         } else if ( outside.getTail() == segment ) {
189             outside.addAllTail( segments );
190         } else
191             throw new IllegalArgumentException( "Segment does not match the last
192             segment on either side of the block.\n" +
193             "Parent: " + segment.getParent().toString() + "\n" +
194             "Inside: " + inside + "\n" +
195             "Outside: " + outside + "\n" +
196             "Segment: " + segment );
197     }
198
199     /**
200     * Swaps the segments on the inside and outside of the block.
201     * @throws IllegalArgumentException If the block is not flippable.
202     */
203     public void flip() {
204         if ( !isFlippable() )
205             throw new IllegalArgumentException( "Trying to flip a block that is not
206             flippable." );
207         final LinkedList<Segment> temp = inside;
208         inside = outside;
209         outside = temp;
210
211         final UnmodifiableLinkedList<Segment> embeddedTemp = embeddedInside;
212         embeddedInside = embeddedOutside;
213         embeddedOutside = embeddedTemp;
214
215         if ( Graph.DEBUG_EMBED ) System.out.println("Block flipped - P = " +
216         parent.toString() + " In " +
217         (inside.isEmpty()?"null":inside.getTail().toString()) + " Out " +
218         (outside.isEmpty()?"null":outside.getTail().toString()));
219     }
220
221     /**
222     * Combines this block with the previous block.
223     * @return Block
224     */
225     public Block concatenate() {
226         previous.flippable &= flippable;
227         previous.inside.addAllTail( inside );
228         previous.outside.addAllTail( outside );
229         previous.embeddedInside.addAllTail( embeddedInside );
230         previous.embeddedOutside.addAllTail( embeddedOutside );
231         parent.removeLastChildBlock();
232         if ( Graph.DEBUG_EMBED ) System.out.println("Block concatenated - P = " +
233         parent.toString() + " In " +
234         (inside.isEmpty()?"null":inside.getTail().toString()) + " Out " +
235         (outside.isEmpty()?"null":outside.getTail().toString()));
236         return previous;
237     }

```

```

225  /**
226   * Fully embeds segments that have a leaf vertex at or above the given index.
227   * @param index
228   * @return boolean – True if the block has been entirely fully embedded,
229   *                 otherwise false.
230   */
231  public boolean embedTo(int index) {
232      while (!inside.isEmpty() && inside.getTail().getLeafVertexDFSIndex() >=
233             index)
234          inside.removeTail();
235      while (!outside.isEmpty() && outside.getTail().getLeafVertexDFSIndex() >=
236             index)
237          outside.removeTail();
238      if (inside.isEmpty() && outside.isEmpty()) {
239          return true;
240      }
241      return false;
242  }
243
244  public void assignBlockToSegments() {
245      for (Segment segment: embeddedInside ) {
246          segment.setSide( INSIDE );
247          segment.setEmbeddedBlock( this );
248      }
249      for (Segment segment: embeddedOutside ) {
250          segment.setSide( OUTSIDE );
251          segment.setEmbeddedBlock( this );
252      }
253  }
254
255  public boolean isRemoved() { return embeddedInside.isEmpty() &&
256      embeddedOutside.isEmpty(); }
257
258  public boolean isInverted() { return inverted; }
259  public void invert() { inverted = !inverted; }
260
261  public String toString() {
262      StringBuffer inBuffer = new StringBuffer();
263      inBuffer.append("[");
264      boolean first = true;
265      for (Segment segment: embeddedInside ) {
266          if ( first ) first = false;
267          else inBuffer.append(", ");
268          inBuffer.append( segment.toString() );
269      }
270      inBuffer.append("]");
271      StringBuffer outBuffer = new StringBuffer();
272      outBuffer.append("[");
273      first = true;
274      for (Segment segment: embeddedOutside ) {
275          if ( first ) first = false;
276          else outBuffer.append(", ");
277          outBuffer.append( segment.toString() );
278      }
279      outBuffer.append("]");
280      StringBuffer buffer = new StringBuffer();
281      buffer.append("{ ");
282      // buffer.append( "Parent: " + parent.toString() + ", " );
283      buffer.append( "In: " );
284      if ( inverted ) {
285          buffer.append( outBuffer );
286          buffer.append( ", Out: " );
287          buffer.append( inBuffer );
288      } else {

```

```

285         buffer.append( inBuffer );
286         buffer.append( " , Out: " );
287         buffer.append( outBuffer );
288     }
289     buffer.append( " }" );
290     return buffer.toString();
291 }
292 }

```

D.2.2 mgtaylor.graph.Edge

```

1 package mgtaylor.graph;

3 import java.util.ArrayList;
4 import java.util.Collection;

6 public class Edge implements GraphObject {

8     /**
9      * The graph the edge belongs to.
10     */
11     private final Graph graph;

13     /**
14      * The vertex designated as the "from" end of the edge.
15     */
16     private final Vertex from;

18     /**
19      * The vertex designated as the "to" end of the edge.
20     */
21     private final Vertex to;

23     /**
24      * The index of the edge within the graph's array of edges.
25     */
26     private final int index;

28     /**
29      * The index of the edge within the "from" vertex's array of connected edges.
30     */
31     private final int fromIndex;

33     /**
34      * The index of the edge within the "to" vertex's array of connected edges.
35     */
36     private final int toIndex;

38     /**
39      * Creates an edge connecting the "from" and "to" vertices.
40      * @param graph - A non-null graph.
41      * @param from - A non-null vertex belonging to the given graph.
42      * @param to - A different non-null vertex belonging to the given graph.
43      * @param index - The index of the edge within the graph's array of edges.
44     */
45     public Edge( final Graph graph, Vertex from, Vertex to, int index ) {
46         if ( graph == null ) throw new IllegalArgumentException(
47             "mgtaylor.graph.Edge( Graph, Vertex, Vertex, int ) - Graph cannot be null."
48         );
49         if ( from == null ) throw new IllegalArgumentException(
50             "mgtaylor.graph.Edge( Graph, Vertex, Vertex, int ) - \"From\" vertex cannot
51             be null." );

```

```

48         if ( to == null )                throw new IllegalArgumentException(
"taylor.graph.Edge( Graph, Vertex, Vertex, int ) - \"To\" vertex cannot be
null." );
49         if ( from.getGraph() != graph ) throw new IllegalArgumentException(
"taylor.graph.Edge( Graph, Vertex, Vertex, int ) - \"From\" vertex belongs
to a different graph." );
50         if ( to.getGraph() != graph ) throw new IllegalArgumentException(
"taylor.graph.Edge( Graph, Vertex, Vertex, int ) - \"To\" vertex belongs
to a different graph." );
51         if ( from == to )                throw new IllegalArgumentException(
"taylor.graph.Edge( Graph, Vertex, Vertex, int ) - \"From\" and \"to\"
vertices cannot be the same vertex." );
52         this.graph = graph;
53         this.from = from;
54         this.to = to;
55         this.index = index;
56         fromIndex = from.connectTo( this );
57         toIndex = to.connectTo( this );
58     }

60     /**
61      * Returns the graph that the edge belongs to.
62      * @return Graph
63      */
64     public Graph getGraph()                { return graph; }

66     /**
67      * Returns the vertex designated as the "From" end.
68      * @return Vertex
69      */
70     public Vertex getFrom()                { return from; }

72     /**
73      * Returns the vertex designated as the "To" end.
74      * @return Vertex
75      */
76     public Vertex getTo()                  { return to; }

78     /**
79      * Returns the index of the edge within the Graph's array of edges.
80      * @return int
81      */
82     public int getIndex()                  { return index; }

84     /**
85      * Checks whether the given vertex is either of the "From" or "To" vertices.
86      * @param vertex - The vertex to check.
87      * @return boolean
88      */
89     public boolean contains( Vertex vertex ) { return vertex == from ||
vertex == to; }

91     /**
92      * Returns the opposite end of the edge from the given vertex or null if the
vertex
93      * is not contained in the edge.
94      * @param vertex
95      * @return Vertex
96      */
97     public Vertex getOtherEndFrom( Vertex vertex ) {
98         if ( vertex == from ) return to;
99         if ( vertex == to ) return from;
100         return null;
101     }

```



```

103  /**
104  * Returns the index to the edge within the given vertex's array of connected
    edges.
105  * @param vertex - A vertex terminating the edge.
106  * @return int
107  */
108  public int getOrderIndex( Vertex vertex ) {
109      if ( vertex == from )
110          return fromIndex;
111      else if ( vertex == to )
112          return toIndex;
113      else
114          throw new IllegalArgumentException( "mtaylor.graph.Edge.getOrderIndex(
    vertex ): Invalid Vertex." );
115  }

118  public String toString() {
119      if ( isDFSVisited() )
120          return "(" + getDFSFrom().toString() + ", " + getDFSTo().toString() +
    ")";
121      return "(" + from.toString() + ", " + to.toString() + ")";
122  }

124  @Override
125  public Collection<Edge> getConnectedEdges() {
126      // TODO Remove
127      Collection<Edge> fe = from.getConnectedEdges();
128      Collection<Edge> te = to.getConnectedEdges();
129      ArrayList<Edge> ce = new ArrayList<Edge>(fe.size() + te.size() );
130      ce.addAll(fe);
131      ce.addAll(te);
132      return ce;
133  }

135  @Override
136  public boolean isConnectedTo(Edge edge) {
137      return this.getFrom().isConnectedTo( edge ) || this.getTo().isConnectedTo(
    edge );
138  }

140  /**
141  * DFS
142  *****/
143  /**
144  * Tag to mark the edge as a stem edge in the DFS tree.
145  */
146  public static final boolean DFS_STEM    = true;

148  /**
149  * Tag to mark the edge as a frond edge in the DFS tree.
150  */
151  public static final boolean DFS_FROND   = !DFS_STEM;

153  /**
154  * Flag to show if the edge has been visited by the DFS algorithm.
155  */
156  private boolean dfsVisited = false;

158  /**
159  * Flag to show if the edge is a Stem or Frond within the DFS tree.
160  */
161  private boolean dfsType    = DFS_FROND;

163  /**

```

```

164     * Sets the flag for whether the edge has been visited by the DFS
165     * algorithm.
166     */
167     public void setDFSVisited( boolean visit )    { dfsVisited = visit; }

169     /**
170     * Checks whether the edge has been visited by the DFS algorithm.
171     * @return boolean
172     */
173     public boolean isDFSVisited()                { return dfsVisited; }

175     /**
176     * Checks whether the edge is a Stem edge within the DFS tree.
177     * @return boolean
178     */
179     public boolean isDFSStem()                   { return isDFSVisited() && dfsType ==
DFS_STEM; }

181     /**
182     * Checks whether the edge is a Frond edge within the DFS tree.
183     * @return boolean
184     */
185     public boolean isDFSFrond()                  { return isDFSVisited() && dfsType ==
DFS_FROND; }

187     /**
188     * Sets the type (Stem or Frond) for the edge within the DFS tree.
189     */
190     public void setDFSType( boolean type )      { dfsType = type; }

192     /**
193     * Returns the vertex the edge originates from within the DFS tree.
194     * If the edge is a Stem edge this will be the vertex with the lower
195     * DFS index; for a Frond edge this will be the vertex with the
196     * higher DFS index.
197     * @return Vertex
198     */
199     public Vertex getDFSFrom() {
200         if ( isDFSStem() )
201             return from.getDFSIndex() < to.getDFSIndex()?from:to;
202         else if ( isDFSFrond() )
203             return from.getDFSIndex() < to.getDFSIndex()?to:from;
204         else
205             return null;
206     }

208     /**
209     * Returns the vertex the edge goes to in the DFS tree.
210     * If the edge is a Stem edge this will be the vertex with the higher
211     * DFS index; for a Frond edge this will be the vertex with the
212     * lower DFS index.
213     * @return Vertex
214     */
215     public Vertex getDFSTo() {
216         if ( isDFSStem() )
217             return from.getDFSIndex() < to.getDFSIndex()?to:from;
218         else if ( isDFSFrond() )
219             return from.getDFSIndex() < to.getDFSIndex()?from:to;
220         else
221             return null;
222     }

224     /*****
225     * Order
226     *****/

```

```

227  /**
228   * The next clockwise edge from this edge connected to the from vertex.
229   */
230  private Edge order_From_Clockwise      = this;

232  /**
233   * The next anti-clockwise edge from this edge connected to the from vertex.
234   */
235  private Edge order_From_AntiClockwise  = this;

237  /**
238   * The next clockwise edge from this edge connected to the to vertex.
239   */
240  private Edge order_To_Clockwise        = this;

242  /**
243   * The next anti-clockwise edge from this edge connected to the to vertex.
244   */
245  private Edge order_To_AntiClockwise    = this;

247  /**
248   * Resets the edge to be unordered.
249   */
250  public void resetOrder() {
251      order_From_Clockwise      = this;
252      order_From_AntiClockwise  = this;
253      order_To_Clockwise        = this;
254      order_To_AntiClockwise    = this;
255  }

257  /**
258   * Checks whether the edge has been ordered about the given vertex.
259   * @param center
260   * @return boolean
261   */
262  public boolean isOrderedAbout( final Vertex center ) {
263      if ( center == from )
264          return from.size() == 1 || (order_From_Clockwise != this &&
265                                     order_From_AntiClockwise != this );
266      // this );
267      else if ( center == to )
268          return to.size() == 1 || (order_To_Clockwise != this &&
269                                   order_To_AntiClockwise != this );
270      // return (order_To_Clockwise != this && order_To_AntiClockwise != this );
271      else
272          throw new IllegalArgumentException( "Center vertex is not connected to
273                                             the edge - " + this + ", " + center );
274  }

276  /**
277   * Gets the edge ordered clockwise from the current edge about the given
278   * central vertex.
279   * @param center
280   * @return Edge
281   */
282  public Edge getOrderClockwiseFrom( final Vertex center ) {
283      if ( center == from )
284          return order_From_Clockwise;
285      else if ( center == to )
286          return order_To_Clockwise;
287      else
288          throw new IllegalArgumentException( "Center vertex is not connected to
289                                             the edge - " + this + ", " + center );
290  }

```

```

288  /**
289   * Gets the edge ordered anti-clockwise from the current edge about the given
290   * central vertex.
291   * @param center
292   * @return Edge
293   */
294  public Edge getOrderAntiClockwiseFrom( final Vertex center ) {
295      if ( center == from )
296          return order_From_AntiClockwise;
297      else if ( center == to )
298          return order_To_AntiClockwise;
299      else
300          throw new IllegalArgumentException(
301              "taylor.graph.Edge.getOrderAntiClockwiseFrom( vertex ): Center vertex
              is not connected to the edge - " + this + ", " + center );
302  }
303
304  /**
305   * Sets the edge ordered clockwise from the current edge about the given
306   * central vertex.
307   * @param center      The vertex at the centre of the edges being ordered.
308   * @param clockwise   The edge being ordered clockwise of this edge.
309   */
310  private void setOrderClockwiseFrom( final Vertex center, final Edge clockwise
311  ) {
312      if ( center == from )
313          order_From_Clockwise = clockwise;
314      else if ( center == to )
315          order_To_Clockwise = clockwise;
316      else
317          throw new IllegalArgumentException( "Center vertex is not connected to
318              the edge - " + this + ", " + center );
319  }
320
321  /**
322   * Sets the edge ordered anti-clockwise from the current edge about the given
323   * central vertex.
324   * @param center      The vertex at the centre of the edges being ordered.
325   * @param antiClockwise The edge being ordered anti-clockwise of this edge.
326   */
327  private void setOrderAntiClockwiseFrom( final Vertex center, final Edge
328  antiClockwise ) {
329      if ( center == from )
330          order_From_AntiClockwise = antiClockwise;
331      else if ( center == to )
332          order_To_AntiClockwise = antiClockwise;
333      else
334          throw new IllegalArgumentException( "Center vertex is not connected to
335              the edge - " + this + ", " + center );
336  }
337
338  /**
339   * Orders the given edge clockwise from the current edge about the
340   * given central vertex.
341   * @param center
342   * @param clockwise
343   */
344  public void setOrderClockwise( final Vertex center, final Edge clockwise ) {
345      Edge cw = getOrderClockwiseFrom( center );
346      cw.setOrderAntiClockwiseFrom( center, clockwise );
347      clockwise.setOrderClockwiseFrom( center, cw );
348      clockwise.setOrderAntiClockwiseFrom( center, this );
349      this.setOrderClockwiseFrom( center, clockwise );
350  }

```

```

347  /**
348   * Orders the given edge anti-clockwise from the current edge about the
349   * given central vertex.
350   * @param center
351   * @param clockwise
352   */
353  public void setOrderAntiClockwise( final Vertex center, final Edge
antiClockwise ) {
354      Edge acw = getOrderAntiClockwiseFrom( center );
355      acw.setOrderClockwiseFrom( center, antiClockwise );
356      antiClockwise.setOrderAntiClockwiseFrom( center, acw );
357      antiClockwise.setOrderClockwiseFrom( center, this );
358      this.setOrderAntiClockwiseFrom( center, antiClockwise );
359  }

361  /**
362   * Planarity Conflict
363   *
364   */
365  /**
366   */
367  private boolean conflict = false;

369  /**
370   * Flags the edge as part of a planarity conflict.
371   */
372  public void setConflictEdge()      { conflict = true; }

374  /**
375   * Checks whether the edge is part of a planarity conflict.
376   * @return True if the edge is part of a planarity conflict, false otherwise.
377   */
378  public boolean hasPlanarityConflict() { return conflict; }
379  }

```

D.2.3 mgtaylor.graph.Graph

```

1  package mgtaylor.graph;

3  import java.util.Arrays;
4  import java.util.Iterator;
5  import java.util.ArrayList;
6  import java.util.regex.Pattern;
7  import java.util.regex.Matcher;
8  import java.io.PrintStream;

10  import mgtaylor.datastructures.LinkList;
11  import mgtaylor.datastructures.UnmodifiableLinkList;

13  /**
14   * Graph is a data structure that contains a list of unique vertices and edges
15   * connecting pairs of vertices.
16   *
17   * @author Martyn G Taylor
18   */
19  public class Graph {
20      public static final boolean DEBUG          = false;
21      public static final boolean DEBUG_DFS      = DEBUG & false;
22      public static final boolean DEBUG_EMBED    = DEBUG & true;
23      public static final boolean DEBUG_ORDER    = DEBUG & true;

```

```

25     private static final String VERTEX_REGEX      = "\\s*[a-zA-Z0-9]+\\s*";
26     private static final String EDGE_REGEX        = "\\s*\\((" + VERTEX_REGEX +
27     "),( " + VERTEX_REGEX + ")\\)\\s*";
27 // private static final Pattern EDGE_PATTERN      = Pattern.compile( "\\s*\\((" +
28 VERTEX_REGEX + "),( " + VERTEX_REGEX + ")\\)\\s*", Pattern.CASE_INSENSITIVE +
29 Pattern.MULTILINE );
28     private static final Pattern EDGE_PATTERN      = Pattern.compile( EDGE_REGEX,
29     Pattern.CASE_INSENSITIVE + Pattern.MULTILINE );
29     private static final String VERTICES_REGEX     =
30     "\\s*nodes\\s*\\{([~\\}]++)\\}\\s*";
30     private static final String EDGES_REGEX        =
31     "\\s*edges\\s*\\{([~\\}]++)\\}\\s*";
31     public static final Pattern VERTICES_PATTERN   = Pattern.compile(
32     VERTICES_REGEX, Pattern.MULTILINE + Pattern.CASE_INSENSITIVE );
32     public static final Pattern EDGES_PATTERN      = Pattern.compile( EDGES_REGEX,
33     Pattern.MULTILINE + Pattern.CASE_INSENSITIVE );
33 // public static final Pattern GRAPH_PATTERN      = Pattern.compile(
34 "\\s*nodes\\s*\\{((" + VERTEX_REGEX + ")*" + VERTEX_REGEX +
35 ")}\\}\\s*edges\\s*\\{((" + EDGE_REGEX + ")*" + EDGE_REGEX + ")}\\}\\s*",
36 Pattern.MULTILINE + Pattern.CASE_INSENSITIVE );
34     public static final Pattern GRAPH_PATTERN      = Pattern.compile(
35     VERTICES_REGEX + EDGES_REGEX, Pattern.MULTILINE + Pattern.CASE_INSENSITIVE );
36
36     protected final ArrayList<Vertex>             vertices;
37     protected final ArrayList<Edge>               edges;
38     private final LinkedList<GraphListener>         graphListeners = new
39     LinkedList<GraphListener>();
39     private String                                 identifier      = "";
40
41     public Graph( int numberOfVertices, int numberOfEdges ) {
42         if ( numberOfVertices < 0 )
43             throw new IllegalArgumentException( "Cannot have negative number of
44             vertices - " + numberOfVertices );
44         if ( numberOfEdges < 0 )
45             throw new IllegalArgumentException( "Cannot have negative number of
46             edges - " + numberOfEdges );
47
48         vertices = new ArrayList<Vertex>( numberOfVertices );
49         edges = new ArrayList<Edge>( numberOfEdges );
50     }
51
52     public Graph(String input) {
53         final Matcher m = GRAPH_PATTERN.matcher(input);
54         if (!m.matches())
55             throw new IllegalArgumentException( "Invalid Graph\n" + input );
56
57         final LinkedList<GraphObject> graphObjects = new LinkedList<GraphObject>();
58         final String[] ids = m.group(1).split(",");
59         final java.util.HashMap<String, Integer> idMap = new
60         java.util.HashMap<String, Integer>( ids.length );
61
62         int index = 0;
63         for ( final String vertexID: ids ) {
64             if ( idMap.containsKey( vertexID.trim() ) )
65                 throw new IllegalArgumentException( "Duplicate Vertex ID\n" + input
66                 );
67             idMap.put( vertexID.trim(), index++ );
68         }
69
70         final int[] vertexSizes = new int[ index ];
71         int e = 0;
72         vertices = new ArrayList<Vertex>( index );
73
74         final String[] edgeStrings = m.group(2).split("(?<=\\))\\s*,\\s*(?=\\()");
75         for ( final String edge: edgeStrings ) {

```

```

73         final Matcher em = EDGE_PATTERN.matcher( edge );
74         if ( em.matches() ) {
75             final String fromID      = em.group(1).trim();
76             final Integer fromIndex = idMap.get( fromID );
77             if ( fromIndex == null )
78                 throw new IllegalArgumentException( "Unknown Edge - From Vertex
              ID - (" + em.group(1).trim() + ", " + em.group(2).trim() + ")\n"
              + input );
80
81             final String toID      = em.group(2).trim();
82             final Integer toIndex  = idMap.get( toID );
83             if ( toIndex == null )
84                 throw new IllegalArgumentException( "Unknown Edge - To Vertex ID
              - (" + em.group(1).trim() + ", " + em.group(2).trim() + ")\n" +
              input );
85
86             if ( fromIndex == toIndex )
87                 throw new IllegalArgumentException( "Invalid Self-Referential
              Edge - (" + em.group(1).trim() + ", " + em.group(2).trim() +
              ")\n" + input );
88
89             vertexSizes[ fromIndex ]++;
90             vertexSizes[ toIndex ]++;
91             e++;
92         } else
93             throw new IllegalArgumentException( "Cannot match input to edge
              pattern: " + edge );
94     }
95
96     for ( final String id: ids ) {
97         final int i = idMap.get( id.trim() );
98         graphObjects.addHead( createVertex( i, id.trim(), vertexSizes[i] ) );
99     }
100
101     edges = new ArrayList<Edge>( e );
102     for ( final String edge: edgeStrings ) {
103         final Matcher em = EDGE_PATTERN.matcher( edge );
104         if ( em.matches() )
105             createEdge( vertices.get( idMap.get( em.group(1).trim() ) ),
106                         vertices.get( idMap.get( em.group(2).trim() ) ) );
107     }
108 }
109
110 /**
111  * Sets the identifier string for the graph.
112  * @param string The identifier for the graph.
113  */
114 public void setIdentifier(String string)    { identifier = (string ==
115 null?"":string); }
116
117 /**
118  * Gets the graph's identifier.
119  * @return String The graph's identifier.
120  */
121 public String getIdentifier()                { return identifier; }
122
123 /**
124  * Gets the vertices of the graph.
125  * @return {@link ArrayList}&lt;Vertex> An ArrayList of Vertices.
126  */
127 public ArrayList<Vertex> getVertices()      { return vertices; }
128
129 /**
130  * Gets the number of vertices in the graph.
131  * @return int The number of vertices.

```

```

130     */
131     public int vertexSize() { return vertices.size(); }

132
133     /**
134     * Checks to see if the given vertex is in this graph.
135     * @param vertex The non-null vertex to check.
136     * @return boolean True if the graph contains the vertex, false otherwise.
137     */
138     public boolean contains( final Vertex vertex ) {
139         if ( vertex == null ) throw new IllegalArgumentException( "Vertex
140         argument cannot be null." );
141         return vertex.getGraph() == this && vertices.get( vertex.getIndex() ) ==
142         vertex;
143     }

144     /**
145     * Gets the edges of the graph.
146     * @return {@link ArrayList}<Edge> An ArrayList of Edges.
147     */
148     public ArrayList<Edge> getEdges() { return edges; }

149     /**
150     * Gets the number of edges in the graph.
151     * @return int The number of edges.
152     */
153     public int edgeSize() { return edges.size(); }

154
155     /**
156     * Checks to see if the given edge is in this graph.
157     * @param vertex The non-null edge to check.
158     * @return boolean True if the graph contains the edge, false otherwise.
159     */
160     public boolean contains( final Edge edge ) {
161         if ( edge == null ) throw new IllegalArgumentException( "Edge argument
162         cannot be null." );
163         return edge.getGraph() == this && edges.get( edge.getIndex() ) == edge;
164     }

165     public void addGraphListener( final GraphListener listener ) {
166         graphListeners.addHead( listener ); }
167     public void removeGraphListener( final GraphListener listener ) {
168         graphListeners.remove( listener ); }
169     public void fireGraphUpdate( final GraphEvent event ) {
170         for ( GraphListener listener : graphListeners )
171             listener.graphChanged( event );
172     }

173     /**
174     * Creates a new vertex with the given label and capacity to connect 0 edges.
175     * @param label The label string for the vertex.
176     * @return Vertex The vertex that has been created.
177     */
178     public Vertex createVertex( String label ) {
179         return createVertex( vertices.size(), label, 0 );
180     }

181     /**
182     * Creates a new vertex with the given label and a maximum initial capacity
183     * to connect edges to the vertex.
184     * @param index
185     * @param label
186     * @param capacity
187     * @return Vertex
188     */

```



```

189     protected Vertex createVertex( final int index, final String label, final int
capacity ) {
190         if ( vertices.size() > index && vertices.get( index ) != null )
191             throw new IllegalArgumentException( "Invalid Index - Vertex already
exists - " + index );
192         if ( label == null )
193             throw new IllegalArgumentException( "Label cannot be null" );
194         Vertex v = new Vertex( this, index, label, capacity );
195         vertices.add( index, v );
196         return v;
197     }

198
199     /**
200     * Creates an Edge joining the two vertices.
201     * @param from Non-null vertex
202     * @param to Non-null vertex
203     * @return Edge
204     */
205     public Edge createEdge(Vertex from, Vertex to) {
206         if ( from == null )
207             throw new IllegalArgumentException( "From vertex is null" );
208         if ( to == null )
209             throw new IllegalArgumentException( "To vertex is null" );
210         Edge edge = new Edge(this, from, to, edges.size());
211         edges.add(edge);
212         return edge;
213     }

214
215     /**
216     * <p>Returns a string representation of the Graph in the format:</p>
217     *
218     * <xmp>nodes { node1, node2, ..., nodeN }
219     * edges
220     * {
221     *   edge1, edge2, ... edgeE
222     * }</xmp>
223     *
224     * <p>Where nodeN, and edgeE are the string representations of the
N<sup>th</sup> node
225     * and E<sup>th</sup> edge respectively.
226     *
227     * @return String
228     */
229     public String toString() {
230         StringBuffer buffer = new StringBuffer();
231         buffer.append("nodes { ");
232         for (Iterator<Vertex> vi = vertices.iterator(); vi.hasNext();) {
233             buffer.append(vi.next().toString());
234             if (vi.hasNext())
235                 buffer.append(", ");
236         }
237         buffer.append(" }\nedges\n{\n ");
238         for (Iterator<Edge> ei = edges.iterator(); ei.hasNext();) {
239             buffer.append(ei.next().toString());
240             if (ei.hasNext())
241                 buffer.append(", ");
242         }
243         buffer.append("\n}");
244         return buffer.toString();
245     }

246
247     /**
248     * DFS
249     *
250     *
251     */
252     private int dfsIndex = 0;

```

```

251     private Vertex dfsRoot = null;

253     /**
254      * Returns the current index used by the DFS algorithm.
255      * @return int
256      */
257     public int getDFSIndex()      { return dfsIndex; }

259     /**
260      * Increments the current index being used by the DFS algorithm.
261      */
262     public void incrementDFSIndex() { dfsIndex++; }

264     /**
265      * Returns the vertex used as the root of the DFS tree.
266      * @return Vertex
267      */
268     public Vertex getDFSRoot()    { return dfsRoot; }

270     /**
271      * Sets the vertex used as the root of the DFS tree.
272      *
273      * @param root The non-null vertex, belonging to this graph, which will be
274      *             used
275      *             as the root of the DFS tree.
276      * @return
277      */
277     public void setDFSRoot( final Vertex root )    {
278         if ( root == null )
279             throw new IllegalArgumentException( "Null vertex passed as root." );
280         if( root.getGraph() == this )
281             dfsRoot = root;
282         else
283             throw new IllegalArgumentException( "Vertex passed as root does not
284             belong to this graph." );
285     }

286     /**
287      * Data structure to link a Vertex and an Edge Iterator for the
288      * Edges connected to that Vertex.
289      * @author Martyn Taylor
290      */
291     private static class DFSVertexEdgesHolder {
292         private final Vertex vertex;
293         private final Iterator<Edge> edges;

295         /**
296          * Creates a holder for the given vertex and the iterator for the
297          * edges connected to that vertex.
298          * @param vertex
299          */
300         private DFSVertexEdgesHolder( final Vertex vertex ) {
301             this.vertex = vertex;
302             this.edges = vertex.getConnectedEdges().iterator();
303         }

305         /**
306          * Gets the vertex.
307          * @return
308          */
309         private Vertex getVertex()      { return vertex; }

311         /**
312          * Gets the iterator for the edges connected to the subject vertex.
313          * @return

```

```

314         */
315     private Iterator<Edge> getEdges() { return edges; }
316 }

318 /**
319  * <p>Sets the DFS root vertex then performs a Depth-First Search</p>
320  *
321  * @param root The non-null vertex, belonging to this graph, which will be
322  * used
323  * as the root of the DFS tree.
324  */
325 public void DFS( final Vertex root ) {
326     setDFSRoot( root );
327     DFS();
328 }

329 /**
330  * <p>Performs a Depth First Search of the graph from the given vertex and
331  * calculates the first and second low points for each vertex.</p>
332  *
333  * <p>The algorithm also generates a list of uni-directional edges connected
334  * to
335  * each vertex, such that edges making up the stem of the DFS tree are
336  * connected
337  * from parent to child and edges that are not part of the stem (frond edges)
338  * are
339  * connected from descendant to ancestor. The algorithm then bucket sorts
340  * these
341  * edges so that they are ordered based on first and second low points as
342  * detailed by Hopcroft and Tarjan.</p>
343  *
344  * @param root The non-null vertex, belonging to this graph, which will be
345  * used
346  * as the root of the DFS tree.
347  */
348 @SuppressWarnings("unchecked")
349 public void DFS() {
350     final Vertex root = getDFSRoot();
351     if ( root == null )
352         throw new IllegalArgumentException( "Root vertex cannot be null." );

353     final UnmodifiableLinkedList<Edge>[] dfsBucket = new UnmodifiableLinkedList[
354         vertexSize() * 2 + 1 ];
355     for ( int i = 0; i < vertexSize() * 2 + 1; i++ )
356         dfsBucket[i] = new UnmodifiableLinkedList<Edge>();

357     for ( Vertex vertex: getVertices() )
358         vertex.resetDFSIndex();

359     for ( Edge edge: getEdges() )
360         edge.setDFSVisited( false );

361     dfsIndex = 0;
362     root.setDFSParent( null );

363     LinkedList<DFSVertexEdgesHolder> edgeIterators = new
364     LinkedList<DFSVertexEdgesHolder>();
365     edgeIterators.addHead( new DFSVertexEdgesHolder( root ) );

366     while ( !edgeIterators.isEmpty() ) {
367         DFSVertexEdgesHolder ve = edgeIterators.getHead();
368         Vertex vertex = ve.getVertex();
369         Iterator<Edge> edgeIterator = ve.getEdges();
370         if ( edgeIterator.hasNext() ) {
371             Edge edge = edgeIterator.next();

```

```

371         if ( !edge.isDFSVisited() ) {
372             edge.setDFSVisited( true );
373             Vertex otherVertex = edge.getOtherEndFrom( vertex );
374             if ( otherVertex.getDFSIndex() == Vertex.DFS_UNSET ) {
375                 edge.setDFSType( Edge.DFS_STEM );
376                 otherVertex.setDFSParent( edge );
377                 edgeIterators.addHead( new DFSVertexEdgesHolder( otherVertex ) );
378             } else {
379                 edge.setDFSType( Edge.DFS_FROND );
380                 if ( otherVertex.getDFSIndex() < vertex.getLowPoint1() ) {
381                     vertex.setLowPoint2( vertex.getLowPoint1() );
382                     vertex.setLowPoint1( otherVertex.getDFSIndex() );
383                 } else if ( vertex.getLowPoint1() < otherVertex.getDFSIndex()
384                     && otherVertex.getDFSIndex() < vertex.getLowPoint2() ) {
385                     vertex.setLowPoint2( otherVertex.getDFSIndex() );
386                 }
387                 dfsBucket[ otherVertex.getDFSIndex() * 2 ].addTail( edge );
388             }
389         } else {
390             Edge parentEdge = vertex.getDFSParent();
391             if ( parentEdge != null ) {
392                 Vertex parent = parentEdge.getOtherEndFrom( vertex );
393                 if ( DEBUG_DFS ) System.out.print( parentEdge + "\tBefore: " +
394                     parent + " (" + parent.getDFSIndex() + ", " +
395                     parent.getLowPoint1() + ", " + parent.getLowPoint2() + ")" + "\t"
396                     + vertex + " (" + vertex.getDFSIndex() + ", " +
397                     vertex.getLowPoint1() + ", " + vertex.getLowPoint2() + ")" );
398                 if ( vertex.getLowPoint1() < parent.getLowPoint1() ) {
399                     parent.setLowPoint2( Math.min( vertex.getLowPoint2(),
400                         parent.getLowPoint1() ) );
401                     parent.setLowPoint1( vertex.getLowPoint1() );
402                 } else if ( vertex.getLowPoint1() == parent.getLowPoint1() ) {
403                     if ( vertex.getLowPoint2() < parent.getLowPoint2() )
404                         parent.setLowPoint2( vertex.getLowPoint2() );
405                 } else if ( vertex.getLowPoint1() < parent.getLowPoint2() ) {
406                     parent.setLowPoint2( vertex.getLowPoint1() );
407                 }
408                 if ( DEBUG_DFS ) System.out.println( "\tAfter: " + parent + " ("
409                     + parent.getDFSIndex() + ", " + parent.getLowPoint1() + ", " +
410                     parent.getLowPoint2() + ")" + "\t" + vertex + " (" +
411                     vertex.getDFSIndex() + ", " + vertex.getLowPoint1() + ", " +
412                     vertex.getLowPoint2() + ")" );
413                 dfsBucket[ vertex.getLowPoint1() * 2 + (vertex.getLowPoint2() <
414                     parent.getDFSIndex() ? 1:0) ].addTail( parentEdge );
415             }
416             edgeIterators.removeHead();
417         }
418     }
419
420     for ( UnmodifiableLinkedList<Edge> orderedEdges: dfsBucket )
421         for ( Edge edge: orderedEdges )
422             edge.getDFSFrom().addDFSAdjacentEdge( edge );
423
424     }
425
426     /**
427     * Segments
428     */
429     private UnmodifiableLinkedList<Segment> segments = null;
430
431     /**
432     * Returns a list of segments.
433     * @return UnmodifiableLinkedList<Segment>
434     */

```

```

424 public UnmodifiableLinkedList<Segment> getSegments() { return segments; }

426 /**
427  * Returns the number of segments.
428  */
429 public int getNumberOfSegments() { return segments.size(); }

431 /**
432  * <p>Generates a list of segments using the DFS adjacency edges.</p>
433  *
434  * <p>{@link Graph#DFS()} must be called before this method.</p>
435  */
436 public void generateSegments() {
437     if ( getDFSRoot() == null )
438         throw new IllegalArgumentException( "DFS Algorithm must have run first"
439         );

440     for ( final Vertex vertex: getVertices() )
441         vertex.setSegment( null );

443     final LinkedList<Iterator<Edge>> edgeIterators = new
444     LinkedList<Iterator<Edge>>();

445     edgeIterators.addHead( getDFSRoot().getDFSAdjacency().iterator() );

447     Segment segment = null;
448     int index = 0;
449     segments = new UnmodifiableLinkedList<Segment>();

451     while( !edgeIterators.isEmpty() ) {
452         Iterator<Edge> edgeIterator = edgeIterators.getHead();
453         if ( edgeIterator.hasNext() ) {
454             Edge edge = edgeIterator.next();
455             if ( !edgeIterator.hasNext() )
456                 edgeIterators.removeHead();
457             if ( segment == null )
458                 segments.addTail( segment = new Segment( index++, edge ) );
459             else if ( segment.continuesStem( edge ) )
460                 segment.addStem( edge );
461             else {
462                 // Handle segments without a cotree edge.
463                 segment.leaf = segment.stem.getTail().getDFSTo();
464                 segments.addTail( segment = new Segment( index++, edge ) );
465             }

467             if ( segment.isComplete() )
468                 segment = null;
469             else
470                 edgeIterators.addHead(
471                     edge.getDFSTo().getDFSAdjacency().iterator() );
472             } else
473                 edgeIterators.removeHead();
474         }
475         if ( segment != null )
476             segment.leaf = segment.stem.getTail().getDFSTo();
477     }

478     /**
479     * Planar Embedding
480     *****/
481     /**
482     * The segment that caused the embedding to fail, or null if the graph is
483     * planar.
484     */
485     private Segment nonPlanarSegment = null;

```

```

487  /**
488   * All the blocks of segments that have been embedded.
489   */
490  private UnmodifiableLinkedList<Block> allBlocks      = null;

492  /**
493   * Checks whether the embedding of the graph succeeded.
494   * @return boolean    True if the graph is planar, false otherwise.
495   */
496  public boolean isPlanar()                { return nonPlanarSegment == null; }

498  /**
499   * Gets a list of all the blocks created during the planar embedding of the
500   * graph.
501   * @return {@link UnmodifiableLinkedList}&lt;Block>    The list of
502   * blocks.
503   */
504  public UnmodifiableLinkedList<Block> getAllBlocks() { return allBlocks; }

506  /**
507   * Adds a block of segments to the graph.
508   * @param block    The non-null block being added to the graph.
509   */
510  public void addBlock( final Block block ) {
511      if ( block == null )
512          throw new IllegalArgumentException( "Cannot add a null block." );
513      allBlocks.addTail( block );
514  }

516  /**
517   * Embeds the segments.
518   */
519  public void embedSegments() {
520      nonPlanarSegment = null;
521      allBlocks         = new UnmodifiableLinkedList<Block>();
522      permutations      = new UnmodifiableLinkedList<SegmentPermutation>();

524      for ( final Segment segment: segments )
525          segment.resetChildIterator();

527      final LinkedList<Segment> segmentIterator = new LinkedList<Segment>();
528      final Segment initialCycle = segments.getHead();
529      initialCycle.setIsEmbedded( true );
530      segmentIterator.addTail( initialCycle );

532      while ( !segmentIterator.isEmpty() ) {
533          Segment segment = segmentIterator.getTail();
534          Segment child = segment.getNextChild();
535          if ( child != null ) {
536              if ( DEBUG_EMBED ) System.out.println( "Processing segment: " +
537                  child + " (child of: " + segment + ")" );

539              if ( !segment.embedChildSegment( child ) ) {
540                  nonPlanarSegment = child;
541                  if ( DEBUG_EMBED ) System.out.println( "Not Embeddable " +
542                      segment.toString() );
543                  break;
544              }
545              if ( child.hasChildSegments() )
546                  segmentIterator.addTail( child );
547          } else {
548              segment.appendDescendantSegments();
549              Segment removed = segmentIterator.removeTail();

```

```

546         if ( DEBUG_EMBED ) System.out.println( "Segment processed: " +
547         segment + " (removed: " + removed + ", remaining: " +
548         segmentIterator + ")" );
549     }
550 }
551
552     for ( final Block block: allBlocks )
553         block.assignBlockToSegments();
554
555     for ( final SegmentPermutation permutation: permutations )
556         permutation.setIsPermutableAboutParent();
557 }
558
559 /*****
560  * Permutations
561  *****/
562
563 /**
564  * The list of segment permutations for the embedded graph.
565  */
566 private UnmodifiableLinkedList<SegmentPermutation> permutations = null;
567
568 /**
569  * Adds a permutation to the graph.
570  * @param permutation The permutation to add to the graph.
571  */
572 public void addPermutation( final SegmentPermutation permutation ) {
573     permutations.addTail( permutation );
574 }
575
576 /**
577  * Gets the permutable segments that are available for the embedded planar
578  * graph.
579  * @return {@link UnmodifiableLinkedList}&lt;{@link SegmentPermutation}&gt; The
580  * list of permutable segments.
581  */
582 public UnmodifiableLinkedList<SegmentPermutation> getPermutations() { return
583     permutations; }
584
585 /*****
586  * Ordering
587  *****/
588
589 /**
590  * Generates the embedding for the graph.
591  */
592 public void generateEdgeSides() {
593     for ( final Segment segment: segments )
594         segment.resetChildIterator();
595     for ( final Edge edge: edges )
596         edge.resetOrder();
597
598     final LinkedList<Segment> segmentIterator = new LinkedList<Segment>();
599     final Segment initialCycle = segments.getHead();
600     Edge ICLeafEdge = null;
601     Edge ICLOutEdge = null;
602     segmentIterator.addTail( initialCycle );
603     initialCycle.setEmbeddedDirection( Segment.CLOCKWISE );
604     for ( final Edge edge: initialCycle.getEdges() ) {
605         if ( edge.isDFSStem() || ( edge.getDFSTo() ==
606             initialCycle.getRootVertex() ) )
607             edge.getDFSTo().setEdgeOrderParent( edge );
608         else
609             ICLeafEdge = edge;
610         if ( Graph.DEBUG_ORDER ) System.out.println( edge.getDFSTo() + " :
611         Parent - " + edge.getDFSTo().toOrderedEdgesString() );
612     }

```

```

603     for ( final Edge edge: initialCycle.getEdges() ) {
604         edge.getDFSFrom().setEdgeOrderStem( edge );
605         if ( ICLeafEdge != null && edge.getDFSFrom() ==
        initialCycle.getLeafVertex() )
606             ICLeafEdge = edge;
607         if ( Graph.DEBUG_ORDER ) System.out.println( edge.getDFSFrom() + ":
        Stem - " + edge.getDFSFrom().toOrderedEdgesString() );
608     }
609     if ( ICLeafEdge != null ) {
610         initialCycle.getLeafVertex().connectClockwiseFrom(ICLeafEdge,
        ICLeafEdge);
611     }

613     while ( !segmentIterator.isEmpty() ) {
614         final Segment segment = segmentIterator.getTail();
615         final Segment child = segment.getNextChild();
616         if ( child == null ) {
617             if ( Graph.DEBUG_ORDER ) System.out.println( "Segment " + segment +
        ": Ordering Complete." );
618             if ( segment.getParent() != null ) {
619                 final UnmodifiableLinkedList<Edge> edges = segment.getEdges();
620                 if ( edges.getTail().isDFSFrom() )
621                     segment.getLeafVertex().orderRemoveEdge( edges.getTail() );
622                 segment.getRootVertex().orderRemoveEdge( edges.getHead() );
623             }
624             segmentIterator.removeTail();
625         } else {
626             if ( Graph.DEBUG_ORDER ) System.out.println( "Segment " + segment +
        ": Ordering Child - " + child );
627             final boolean direction = segment.getEmbeddingDirection();
628             final boolean childDirection = child.getSide() ==
        Block.INSIDE?direction:!direction;
629             final UnmodifiableLinkedList<Edge> childEdges = child.getEdges();
630             final Vertex childRoot = child.getRootVertex();
631             final Vertex childLeaf = child.getLeafVertex();
632             child.setEmbeddedDirection( childDirection );

634             for ( final Edge edge: childEdges )
635                 if ( edge.isDFSStem() ) {
636                     edge.getDFSFrom().setEdgeOrderParent( edge );
637                     if ( Graph.DEBUG_ORDER ) System.out.println( edge.getDFSFrom() +
        ": Parent - " + edge.getDFSFrom() + " - " +
        edge.getDFSFrom().toOrderedEdgesString() );
638                 }
639             for ( final Edge edge: childEdges )
640                 if ( edge.getDFSFrom() != childRoot ) {
641                     edge.getDFSFrom().setEdgeOrderStem( edge );
642                     if ( Graph.DEBUG_ORDER ) System.out.println( edge.getDFSFrom()
        + ": Stem - " + edge.getDFSFrom() + " - " +
        edge.getDFSFrom().toOrderedEdgesString() );
643                 }

645             segment.setChildEmbeddedDirection( childDirection );

647             childRoot.orderOutboundEdge( childEdges.getHead() );
648             final Edge childTail = childEdges.getTail();
649             if ( childTail.isDFSFrom() )
650                 childLeaf.orderInboundEdge( childTail );

652             segmentIterator.addTail( child );
653         }
654     }
655 }

657 /**

```



```

658     * Returns a string representation of the cyclic order that vertices
659     * are connected.
660     * @return String
661     */
662     public String toVertexCyclicOrderString() {
663         final StringBuilder sb = new StringBuilder();
664         for ( final Vertex vertex : getVertices() ) {
665             sb.append( vertex.toString() );
666             sb.append( ": " );
667             sb.append( vertex.toOrderedEdgesString() );
668             sb.append( '\n' );
669         }
670         return sb.toString();
671     }

672
673     /*****
674     * Statistics
675     *****/

676
677     /**
678     * Calculates the average segment depth for the graph.
679     * @return double The average segment depth.
680     */
681     public double calculateAverageSegmentDepth() {
682         int depthTotal = 0;
683         for ( final Segment segment: segments ) {
684             segment.setDepthFromParent();
685             depthTotal += segment.getDepth();
686         }
687         return (double) depthTotal / segments.size();
688     }

689
690     /**
691     * Calculates the maximum segment depth for the graph.
692     * @return int The maximum segment depth.
693     */
694     public int calculateMaximumSegmentDepth() {
695         int maxDepth = 0;
696         for ( final Segment segment: segments ) {
697             segment.setDepthFromParent();
698             if ( segment.getDepth() > maxDepth )
699                 maxDepth = segment.getDepth();
700         }
701         return maxDepth;
702     }

703
704     /**
705     * Calculates the total number of conflicts.
706     * @return int The maximum segment depth.
707     */
708     public int calculateTotalConflicts() {
709         int conflicts = 0;
710         for ( final Segment segment: segments )
711             conflicts += segment.getConflictingSegments().size();
712         // if ( conflicts != segments.size() - 3 ) System.err.println(
713         segments.size() + ", " + conflicts );
714         return conflicts / 2;
715     }

716
717     public int calculateNumberOfBlocks() {
718         int numBlocks = 0;
719         for ( final Block block : allBlocks ) {
720             if ( block.hasNoSegmentsInside() && block.hasNoSegmentsOutside() )
721                 continue;
722             numBlocks++;
723         }
724     }

```

```

722     }
723     return numBlocks;
724 }

726 public int calculateNumberOfFlippableBlocks() {
727     int numBlocks = 0;
728     for ( final Block block : allBlocks ) {
729         if ( block.hasNoSegmentsInside() && block.hasNoSegmentsOutside() )
730             continue;
731         if ( block.isFlippable() )
732             numBlocks++;
733     }
734     return numBlocks;
735 }

737 /**
738  * Each segment is only added to an existing block if there is a conflict ,
739  * therefore the total number of conflicts indicates the size of each block
740  * (this should be equal to number of segments - 1 - )
741  * @return
742  */
743 public double calculateAverageBlockSize() {
744     final int nb = calculateNumberOfBlocks();
745     final int tc = calculateTotalConflicts();
746     return (nb == 0?0:(((double)tc)/nb + 1));
747 }

749 /**
750  * Calculates the average of a segment's depth multiplied by the number of
751  * children it has.
752  * @return double
753  */
754 public double calculateAverageSegmentDepthXChildren() {
755     int depthTotal = 0;
756     for ( final Segment segment: segments ) {
757         segment.setDepthFromParent();
758         depthTotal += (segment.getDepth() + 1 ) *
759             (segment.getChildSegments().size() + 1);
760     }
761     return (double) depthTotal / segments.size();
762 }

763 public void printDepthsForAllRootVertices() {
764     for ( final Vertex root: vertices ) {
765         DFS( root );
766         generateSegments();
767         System.out.println( root.toString() + '\t' +
768             calculateAverageSegmentDepth() );
769     }
770 }

771 public void printDepthsForAllRootVertices( final int start, final int end ) {
772     for ( int i = start; i <= end && i <= vertices.size(); i++ ) {
773         final Vertex root = vertices.get( i );
774         DFS( root );
775         generateSegments();
776         System.out.println( root.toString() + '\t' +
777             calculateAverageSegmentDepth() );
778     }
779 }

780 public void printDepthsXChildrenForAllRootVertices() {
781     for ( final Vertex root: vertices ) {
782         DFS( root );
783         generateSegments();

```

```

783         System.out.println( root.toString() + '\t' +
784             calculateAverageSegmentDepthXChildren() );
785     }

787     public void printEdgeStandardDeviation() {
788         int sum = 0;
789         int sumsq = 0;
790         for (Vertex v: getVertices() ) {
791             int size = v.getConnectedEdges().size();
792             sum += size;
793             sumsq += size*size;
794         }

796         double std = Math.sqrt((double) sumsq/vertexSize() - (double)
            (sum*sum)/(vertexSize()*vertexSize()));

798         System.out.println( vertexSize() + "\t" + std );
799     }

801     public void printVertexSizes() {
802         java.util.HashMap<Integer, Integer> count = new java.util.HashMap<Integer,
            Integer>();
803         for (Vertex v: getVertices() ) {
804             int size = v.getConnectedEdges().size();
805             if ( count.containsKey( size ) ) {
806                 Integer c = count.remove( size );
807                 count.put( size, c + 1 );
808             } else
809                 count.put( size, 1 );
810         }
811         int[] orderedCounts = new int[ count.size() ];
812         int i = 0;
813         for( int c: count.keySet() )
814             orderedCounts[i++] = c;
815         Arrays.sort( orderedCounts );
816         int[] counts = new int[ count.size() ];
817         i = 0;
818         for( int c: orderedCounts ) {
819             counts[i++] = count.get( c );
820         }

822         for ( i = 0; i < counts.length; i++ )
823             System.out.println( vertexSize() + "\t" + orderedCounts[i] + "\t" +
            counts[i] );
824     }

826     /*****
827     * Speed Testing
828     *****/

830     /**
831     *
832     */
833     public static final String VERTEX_STRING      = "Vertices";
834     public static final String EDGE_STRING        = "Edges";
835     public static final String ROOT_STRING        = "Root";
836     public static final String START_STRING       = "Timing Started";
837     public static final String DFS_STRING         = "DFS Completed";
838     public static final String GENERATION_STRING  = "Segment Generation Completed";
839     public static final String EMBEDDING_STRING   = "Segment Embedding
            Completed";
840     public static final String EDGE_SIDES_STRING  = "Edge Sides Generated";

842     public void timePlanarityTest() {

```

```

843         if ( vertexSize() > 0 )
844             timePlanarityTest( getVertices().get( 0 ), System.out );
845     }

847     public void timePlanarityTest( final Vertex root ) {
848         timePlanarityTest( root, System.out );
849     }

851     public void timePlanarityTest( final PrintStream writer ) {
852         if ( vertexSize() > 0 )
853             timePlanarityTest( getVertices().get( 0 ), writer );
854     }

856     private static final boolean SHOW_COMPILATION_TIMES = false;
857     private static final boolean SHOW_CLASS_LOADING = false;

859     private static final java.lang.management.CompilationMXBean compMXBean =
SHOW_COMPILATION_TIMES?java.lang.management.ManagementFactory.getCompilationMXBean():null;
860     private static final java.lang.management.ClassLoadingMXBean classMXBean =
SHOW_CLASS_LOADING?java.lang.management.ManagementFactory.getClassLoadingMXBean():null;

862     public void silentPlanarityTest( final Vertex root ) {
863         setDFSRoot( root );
864         silentPlanarityTest();
865     }
866     public void silentPlanarityTest() {
867         DFS();
868         generateSegments();
869         embedSegments();
870         generateEdgeSides();
871     }

873     public void timePlanarityTest( final Vertex root, final PrintStream writer ) {
874         if ( writer == null ) throw new IllegalArgumentException( "Output stream
cannot be null" );

876         if ( root != null )
877             setDFSRoot( root );
878         writer.println( ROOT_STRING + ": " + getDFSRoot().toString() );

880         final long classes =
SHOW_CLASS_LOADING?classMXBean.getTotalLoadedClassCount():0;
881         final long unloaded =
SHOW_CLASS_LOADING?classMXBean.getUnloadedClassCount():0;
882         long startCompTime = 0;
883         long doneCompTime = 0;

885         writer.println( START_STRING );
886         if ( SHOW_COMPILATION_TIMES ) startCompTime =
compMXBean.getTotalCompilationTime();
887         long start = System.nanoTime();
888         DFS();
889         long done = System.nanoTime();
890         if ( SHOW_COMPILATION_TIMES ) {
891             doneCompTime = compMXBean.getTotalCompilationTime();
892             writer.println( DFS_STRING + ": " + (done - start) + " - " +
((doneCompTime - startCompTime) * 1000000) + " = " + (done - start -
(doneCompTime - startCompTime) * 1000000) );
893         } else
894             writer.println( DFS_STRING + ": " + (done - start) );

896         if ( SHOW_COMPILATION_TIMES ) startCompTime =
compMXBean.getTotalCompilationTime();
897         start = System.nanoTime();
898         generateSegments();

```

```

899     done = System.nanoTime();
900     if ( SHOW_COMPILATION_TIMES ) {
901         doneCompTime = compMXBean.getTotalCompilationTime();
902         writer.println( GENERATION_STRING + ": " + (done - start) + " - " +
            ((doneCompTime - startCompTime) * 1000000) + " = " + (done - start -
            (doneCompTime - startCompTime) * 1000000) );
903     } else
904         writer.println( GENERATION_STRING + ": " + (done - start) );

906     if ( SHOW_COMPILATION_TIMES ) startCompTime =
        compMXBean.getTotalCompilationTime();
907     start = System.nanoTime();
908     embedSegments();
909     done = System.nanoTime();
910     if ( SHOW_COMPILATION_TIMES ) {
911         doneCompTime = compMXBean.getTotalCompilationTime();
912         writer.println( EMBEDDING_STRING + ": " + (done - start) + " - " +
            ((doneCompTime - startCompTime) * 1000000) + " = " + (done - start -
            (doneCompTime - startCompTime) * 1000000) );
913     } else
914         writer.println( EMBEDDING_STRING + ": " + (done - start) );

916     if ( SHOW_COMPILATION_TIMES ) startCompTime =
        compMXBean.getTotalCompilationTime();
917     start = System.nanoTime();
918     generateEdgeSides();
919     done = System.nanoTime();
920     if ( SHOW_COMPILATION_TIMES ) {
921         doneCompTime = compMXBean.getTotalCompilationTime();
922         writer.println( EDGE_SIDES_STRING + ": " + (done - start) + " - " +
            ((doneCompTime - startCompTime) * 1000000) + " = " + (done - start -
            (doneCompTime - startCompTime) * 1000000) );
923     } else
924         writer.println( EDGE_SIDES_STRING + ": " + (done - start) );

926     if ( SHOW_CLASS_LOADING ) {
927         final long doneClasses = classMXBean.getTotalLoadedClassCount();
928         final long doneUnloaded = classMXBean.getUnloadedClassCount();
929         writer.println( "Classes Loaded: " + (doneClasses - classes) + ",
            Classes Unloaded: " + (doneUnloaded - unloaded) );
930     }
931     // writer.println( getSegments().getHead().toIndentedString( "", " " ) );
932     /* for ( final Vertex vertex: getVertices() ) {
933         writer.print( vertex + ": [" );
934         boolean first = true;
935         for ( final Edge edge: vertex.getEdgeOrder() ) {
936             if ( first )
937                 first = false;
938             else
939                 writer.print( ", " );
940             writer.print( edge.getOtherEndFrom( vertex ) );
941         }
942         writer.println( "]" );
943     }
944     /**/
945 }

948 public static void main( String[] args ) {
949     Graph g;
950     // g = new Graph( "nodes { 0, A, B, C, D, E, F, G, H, I, J }\n" +
951     // "edges { (0, A), (0, B), (0, C), (0, D), (0, E), (0, F), (0, G), (0,
        H), (0, I), (0, J) } " );

```

```

952 //      g = new Graph( "nodes { A, B, C, D, E, F, G, H }\nedges { (A,B), (A,C),
(A,D), (A,E), (A,F), (A,G), (B,C), (C,D), (C,F), (D,E), (D,G), (E,F), (E,G),
(C,H), (H,A) }" );
953      int size = 5000;
954      g = new mgtaylor.graph.graphTypes.PlanarMultiWheelGraph(size, 3, true);
955      for ( int i = 0; i < size; i++ )
956          g.timePlanarityTest( g.getVertices().get( i ) );

958      System.out.println( g );
959      for ( Vertex vertex: g.getVertices() )
960          System.out.println( vertex + "\t" + vertex.getDFSIndex() + "\t" +
vertex.getLowPoint1() + "\t" + vertex.getLowPoint2() + "\t" +
vertex.getDFSAdjacency() );
961      System.out.println( "\t" + g.getSegments().toString( ",\n\t" ) );
962  }
963  /**/
964  }

```

D.2.4 mgtaylor.graph.GraphEvent

```

1  package mgtaylor.graph;

3  import java.util.*;

5  public class GraphEvent {
6      public static final int NEW_GRAPH      = 1;
7      public static final int NODE_ADDED     = 2;
8      public static final int NODE_DELETED   = 4;
9      public static final int NODE_EDITED    = 8;
10     public static final int EDGE_ADDED      = 16;
11     public static final int EDGE_DELETED    = 32;
12     public static final int EDGE_EDITED     = 64;

14     private final Collection<GraphObject> changedItems;

16     private final int status;

18     public GraphEvent( final int status, final GraphObject object ) {
19         this.status = status;
20         changedItems = new ArrayList<GraphObject>( 1 );
21         changedItems.add( object );
22     }

24     public GraphEvent( final int status, final Collection<GraphObject> objects ) {
25         this.status = status;
26         changedItems = objects;
27     }

29     public int getStatus() { return status; }
30     public Collection<GraphObject> getChangedItems() { return changedItems; }
31     public Collection<Vertex> getChangedNodes() {
32         LinkedList<Vertex> nodes = new LinkedList<Vertex>();
33         for ( GraphObject object: changedItems )
34             if ( object instanceof Vertex )
35                 nodes.add( (Vertex) object );
36         return nodes;
37     }
38     public Collection<Edge> getChangedEdges() {
39         LinkedList<Edge> edges = new LinkedList<Edge>();
40         for ( GraphObject object: changedItems )
41             if ( object instanceof Edge )
42                 edges.add( (Edge) object );

```

```

43         return edges;
44     }
45 }

```

D.2.5 mgtaylor.graph.GraphListener

```

1 package mgtaylor.graph;

3 public interface GraphListener {
4     public void graphChanged(GraphEvent event);
5 }

```

D.2.6 mgtaylor.graph.GraphObject

```

1 package mgtaylor.graph;

3 import java.util.Collection;

5 public interface GraphObject {
6     // public boolean isConnectedTo(Vertex vertex);
7     public boolean isConnectedTo(Edge edge);
8     // public Collection<Vertex> getConnectedVertices();
9     public Collection<Edge> getConnectedEdges();
10    public String toString();
11 }

```

D.2.7 mgtaylor.graph.Path

```

1 package mgtaylor.graph;

3 import java.util.Iterator;

5 import mgtaylor.datastructures.UnmodifiableLinkedList;

7 public class Path extends PathSegment {

9     /**
10      * The root (first) vertex of the segment.
11      */
12     protected final Vertex root;

14     /**
15      * The leaf (last) vertex of the segment.
16      */
17     protected Vertex leaf = null;

19     /**
20      * Creates a segment (or the start of a segment) from the given edge.
21      * The segment will be complete once a frond edge has been added.
22      * @param index      A unique sequential identifier for the Segment.
23      * @param edge       The first edge of the segment.
24      */
25     public Path( final Edge edge ) {
26         addStem( edge );

```

```

28         this.root    = edge.getDFSFrom();
29     }

30
31     /**
32     * Returns the graph containing the segment.
33     * @return Graph The graph containing the segment.
34     */
35     public Graph getGraph()                { return root.getGraph(); }

36
37     /**
38     * Returns the root vertex of the segment; this is the first vertex in the
39     * sequence of connected unidirectional DFS edges.
40     * @return Vertex The root vertex.
41     */
42     @Override
43     public Vertex getRootVertex()          { return root; }

44
45     /**
46     * Returns the DFS index of the root vertex.
47     * @return int The DFS index of the root vertex.
48     */
49     @Override
50     public int getRootVertexDFSIndex()      { return root.getDFSIndex(); }

51
52     /**
53     * Returns the leaf vertex of the segment; this is the last vertex in the
54     * sequence of connected unidirectional DFS edges.
55     * @return Vertex The leaf vertex.
56     */
57     @Override
58     public Vertex getLeafVertex()          { return leaf; }

59
60     /**
61     * Returns the DFS index of the leaf vertex.
62     * @return int The DFS index of the leaf vertex.
63     */
64     @Override
65     public int getLeafVertexDFSIndex()      { return leaf.getDFSIndex(); }

66
67     /**
68     * Returns the segment that the root vertex belongs to (or null if the
69     * root vertex is the root vertex of the initial cycle).
70     * @return
71     */
72     @Override
73     public Segment getParent() {
74         final Segment segment = root.getSegment();
75         if ( segment.getRootVertex() == root )
76             return null;
77         return segment;
78     }

79
80     /**
81     * The second lowest DFS index reachable from any vertex on the stem of this
82     * segment. If the segment
83     * is a single frond edge then this defaults to the root vertex's DFS index;
84     * otherwise it is the
85     * low point 2 index for the first vertex of the stem (after the root vertex).
86     * @return int The DFS index of the second low point vertex.
87     */
88     @Override
89     public int getLowPoint2Index()          {
90         if (hasStem())
91             return stem.getHead().getDFSTo().getLowPoint2();
92         return getRootVertexDFSIndex();
93     }

```



```

91     }

92
93     /**
94     * Adds the edge to the stem of the segment.
95     * @param edge
96     */
97     @Override
98     public void addStem( final Edge edge ) {
99         super.addStem( edge );

101         if ( edge.isDFSFrond() )
102             leaf = edge.getDFSTo();
103         else if ( !edge.isDFSStem() )
104             throw new IllegalArgumentException( "Edge has not been visited by DFS
105             algorithm: " + edge );
106     }

107
108     /**
109     * Checks to see if the segment is fully formed.
110     * @return boolean
111     */
112     public boolean isComplete() { return leaf != null; }

113
114     /**
115     * Gets the path from the root vertex to the second low point vertex.
116     * @return
117     */
118     public Path getPathToLowPoint2() {
119         final int lowPt2 = getLowPoint2Index();
120         Edge e = stem.getHead();
121         Vertex to = e.getDFSTo();
122         Path path = new Path( e );

123         while ( to.getDFSIndex() != lowPt2 ) {
124             final Iterator<Edge> edges = to.getDFSAdjacency().iterator();
125             to = (e = edges.next()).getDFSTo();
126             while ( to.getLowPoint1() != lowPt2 && to.getLowPoint2() != lowPt2 )
127                 to = (e = edges.next()).getDFSTo();
128             path.addStem( e );
129         }
130         return path;
131     }

132
133     /**
134     * Checks whether this and the given path (that must both be attached to the
135     * same
136     * segment) conflict when only considering the paths to their leaf and low
137     * point 2
138     * vertices.
139     * @param path
140     * @return
141     */
142     public boolean conflictsWith( final Path path ) {
143         if ( this.getParent() != path.getParent() )
144             throw new IllegalArgumentException( "Paths do not have the same
145             segment." );
146         if ( this.getRootVertexDFSIndex() < path.getRootVertexDFSIndex() ) {
147
148         } else if ( this.getRootVertexDFSIndex() > path.getRootVertexDFSIndex() ) {
149
150         } else {
151             if ( this.getLeafVertexDFSIndex() < path.getLeafVertexDFSIndex() ) {
152                 return this.getLowPoint2Index() > path.getLeafVertexDFSIndex();
153             }
154         }
155     }

```

```

151         } else if ( this.getLeafVertexDFSIndex() > path.getLeafVertexDFSIndex()
152         ) {
153             return path.getLowPoint2Index() > this.getLeafVertexDFSIndex();
154         } else {
155             return this.getLowPoint2Index() < this.getRootVertexDFSIndex() &&
156                    path.getLowPoint2Index() < path.getRootVertexDFSIndex();
157         }
158     }
159     return false;
160 }
161 /**
162  * Gets the path to Low Point 2 vertex starting from the deepest possible
163  * vertex on the stem.
164  * @return
165  */
166 public Path getPathStubToLowPoint2() {
167     final int lowPt2 = getLowPoint2Index();
168     Edge e = stem.getHead();
169     Vertex to = e.getDFSTo();
170     Path path = null;
171     boolean hasBranched = false;
172
173     while ( to.getDFSIndex() != lowPt2 ) {
174         final Iterator<Edge> edges = to.getDFSAdjacency().iterator();
175         boolean firstEdge = true;
176         to = (e = edges.next()).getDFSTo();
177         while ( to.getLowPoint1() != lowPt2 && to.getLowPoint2() != lowPt2 ) {
178             to = (e = edges.next()).getDFSTo();
179             firstEdge = false;
180         }
181         if ( !firstEdge && !hasBranched ) {
182             path = new Path( e );
183             hasBranched = true;
184         } else if ( hasBranched )
185             path.addStem( e );
186     }
187     return path;
188 }
189
190 public Path extendPathBackTo( final Segment segment ) {
191     Vertex r = root;
192     final UnmodifiableLinkedList<Edge> edges = new UnmodifiableLinkedList<Edge>();
193     while ( r.getSegment() != segment ) {
194         final Edge e = r.getDFSParent();
195         edges.addHead( e );
196         r = e.getDFSFrom();
197     }
198     Path path = null;
199     for ( final Edge e: edges )
200         if ( path == null )
201             path = new Path( e );
202         else
203             path.addStem( e );
204     for ( final Edge e: stem )
205         if ( path == null )
206             path = new Path( e );
207         else
208             path.addStem( e );
209     return path;
210 }

```

D.2.8 mgtaylor.graph.PathSegment

```

1 package mgtaylor.graph;

2
3 import mgtaylor.datastructures.LinkList;
4 import mgtaylor.datastructures.UnmodifiableLinkList;

5
6 public class PathSegment {
7     /**
8      * The list of edges that comprise the segment.
9      */
10    protected final UnmodifiableLinkList<Edge> stem = new
        UnmodifiableLinkList<Edge>();

11
12    /**
13     * Creates a segment (or the start of a segment) from the given edge.
14     * The segment will be complete once a frond edge has been added.
15     */
16    public PathSegment() {}

17
18    /**
19     * Returns the root vertex of the segment; this is the first vertex in the
20     * sequence of connected unidirectional DFS edges.
21     * @return Vertex The root vertex.
22     */
23    public Vertex getRootVertex() { return
        stem.isEmpty() ? null : stem.getHead().getDFSFrom(); }

24
25    /**
26     * Returns the DFS index of the root vertex.
27     * @return int The DFS index of the root vertex.
28     */
29    public int getRootVertexDFSIndex() { return
        stem.isEmpty() ? -1 : stem.getHead().getDFSFrom().getDFSIndex(); }

30
31    /**
32     * Returns the leaf vertex of the segment; this is the last vertex in the
33     * sequence of connected unidirectional DFS edges.
34     * @return Vertex The leaf vertex.
35     */
36    public Vertex getLeafVertex() { return
        stem.isEmpty() ? null : stem.getTail().getDFSTo(); }

37
38    /**
39     * Returns the DFS index of the leaf vertex.
40     * @return int The DFS index of the leaf vertex.
41     */
42    public int getLeafVertexDFSIndex() { return
        stem.isEmpty() ? -1 : stem.getTail().getDFSTo().getDFSIndex(); }

43
44    /**
45     * Returns the segment that the root vertex belongs to (or null if the
46     * root vertex is the root vertex of the initial cycle).
47     * @return
48     */
49    public Segment getParent() {
50        final Vertex root = getRootVertex();
51        final Segment segment = root.getSegment();
52        if ( segment.getRootVertex() == root )
53            return null;
54        return segment;
55    }

56    /**

```

```

58      * The second lowest DFS index reachable from any vertex on the stem of this
59      * segment. If the segment
60      * is a single frond edge then this defaults to the root vertex's DFS index;
61      * otherwise it is the
62      * low point 2 index for the first vertex of the stem (after the root vertex).
63      * @return int    The DFS index of the second low point vertex.
64      */
65      public int getLowPoint2Index() {
66          if (hasStem())
67              return stem.getHead().getDFSTo().getLowPoint2();
68          return getRootVertexDFSIndex();
69      }
70
71      /**
72      * Gets the last vertex in the stem.
73      * @return
74      */
75      public Vertex getLastStemVertex() { return
76          stem.getTail().getDFSFrom(); }
77
78      public boolean continuesStem( final Edge edge ) {
79          if ( edge == null )
80              throw new IllegalArgumentException( "Invalid null edge" );
81          return ( stem.isEmpty() || edge.getDFSFrom() == stem.getTail().getDFSTo() );
82      }
83
84      /**
85      * Adds the edge to the stem of the segment.
86      * @param edge
87      */
88      public void addStem( final Edge edge ) {
89          if ( !continuesStem( edge ) )
90              throw new IllegalArgumentException( "Edge does not connect to most
91              recent stem edge: " + stem.getTail() + "->" + edge );
92
93          stem.addTail( edge );
94      }
95
96      /**
97      * Checks whether the segment has a stem and is not just a single frond edge.
98      * @return
99      */
100      public boolean hasStem() { return
101          stem.getHead().isDFSStem(); }
102
103      /**
104      * Gets the edges
105      */
106      public UnmodifiableLinkedList<Edge> getEdges() { return stem; }
107
108      /**
109      * Gets the stem vertices for the segment.
110      * @return {@link LinkedList}&lt;{@link Vertex}> The vertices comprising
111      * the stem of the segment.
112      */
113      public UnmodifiableLinkedList<Vertex> getStemVertices() {
114          final UnmodifiableLinkedList<Vertex> stemVertices = new
115          UnmodifiableLinkedList<Vertex>();
116          for ( Edge edge: stem )
117              if ( edge != stem.getTail() )
118                  stemVertices.addTail( edge.getDFSTo() );
119          return stemVertices;
120      }
121
122      }
123
124      }

```

D.2.9 mgtaylor.graph.Segment

```

1 package mgtaylor.graph;
2
3 import java.util.Iterator;
4
5 import mgtaylor.datastructures.UnmodifiableLinkedList;
6 import mgtaylor.datastructures.LinkList;
7
8 public class Segment extends Path {
9     public static final boolean DEBUG_SET_SIDES          = false;
10    public static final boolean DEBUG_ORDER              = false;
11
12    public static final boolean CLOCKWISE                 = true;
13    public static final boolean ANTICLOCKWISE            = !CLOCKWISE;
14
15    /**
16     * A unique sequential identifier for the segment.
17     */
18    private final int index;
19
20    /**
21     * The parent segment.
22     * The segment that contains the root vertex in it's stem.
23     */
24    private final Segment parent;
25
26    /**
27     * The previous segment.
28     * The previous child segment of the same parent; or if the parent has no
29     * children
30     * then the parent segment.
31     */
32    private final Segment previous;
33
34    /**
35     * Creates a segment (or the start of a segment) from the given edge.
36     * The segment will be complete once a frond edge has been added.
37     * @param index      A unique sequential identifier for the Segment.
38     * @param edge       The first edge of the segment.
39     */
40    public Segment( int index, Edge edge ) {
41        super( edge );
42
43        this.index = index;
44        this.parent = root.getSegment();
45
46        if ( parent == null ) {
47            root.setSegment( this );
48            previous = null;
49        } else {
50            previous =
51                parent.getChildSegments().isEmpty()?parent:parent.getChildSegments().getTail();
52            parent.addChildSegment( this );
53        }
54    }
55
56    /**
57     * Returns the unique index for the segment.
58     * @return

```

```

57     */
58     public int getSegmentIndex()                { return index; }

60     /**
61     * The parent segment is the segment which first contains the segment's root
62     * vertex in it's
63     * stem; the exception for this is the initial cycle which has a null parent
64     * element otherwise the
65     * parent would be self-referential.
66     * @return Segment The segment that is the parent for this segment or null
67     * in the case of the
68     * initial cyclic segment's parent.
69     */
70     public Segment getParent()                    { return parent; }

71     /**
72     * The previous segment is the segment prior to this one in the parent's list
73     * of child segments; if
74     * this is the first child of the parent then the previous segment is the
75     * parent and if this
76     * segment is the initial cycle then the previous segment is null.
77     * @return Segment The previous segment.
78     */
79     public Segment getPrevious()                  { return previous; }

80     /**
81     * The segment which first contains this segment's leaf vertex.
82     * @return
83     */
84     public Segment getLeafSegment()               { return leaf.getSegment(); }

85     /**
86     * Adds the edge to the stem of the segment.
87     * @param edge
88     */
89     public void addStem( final Edge edge ) {
90         super.addStem( edge );
91
92         if ( edge.isDFSStem() ) {
93             edge.getDFSTo().setSegment( this );
94         }
95     }

96     /**
97     * Child Segments
98     */
99     /**
100    * The segments that have root vertices on this segment's stem.
101    */
102    private final UnmodifiableLinkedList<Segment> childSegments = new
    UnmodifiableLinkedList<Segment>();

103
104    /**
105    *
106    */
107    private Iterator<Segment> childIterator = null;

108
109    /**
110    *
111    */
112    private Iterator<Segment> permutationIterator = null;

113
114    /**
115    * Adds a child segment to this (the parent's) list of child segments.

```

```

116     * @param segment
117     */
118     protected void addChildSegment( Segment segment ) {
119         childSegments.addTail( segment );
120     }

122     /**
123     * Checks whether the segment has children.
124     * @return
125     */
126     public boolean hasChildSegments() {
127         return !childSegments.isEmpty();
128     }

130     /**
131     * Returns the list of child segments.
132     * @return {@link LinkList}&lt;{@link Segment}&gt;;
133     */
134     public UnmodifiableLinkList<Segment> getChildSegments() {
135         return childSegments;
136     }

138     /**
139     * Resets the iterator over the child segments.
140     */
141     public void resetChildIterator() {
142         childIterator = childSegments.iterator();
143         permutationIterator = null;
144     }

146     /**
147     * Gets the next child from a stored iterator over the list of child segments.
148     * <p>
149     * This method is first called by {@link Graph#embedSegments()} and the
150     * segments
151     * will not have any permutations set and they will be visited in the order
152     * they
153     * were generated; it is during that method invocation when segments will be
154     * added to
155     * permutation groups, if required.
156     * <p>
157     * When this method is called by {@link Graph#generateEdgeSides()}, if the
158     * segments
159     * are not members of a permutable group then they will be visited in the
160     * order in
161     * which they were generated. If the segments are part of a permutable group
162     * then
163     * they will be visited in the current permutation order and once all the
164     * segments
165     * in the permutation are visited then the visiting order will return to the
166     * next
167     * permutation in the default order (assuming that is not also part of a
168     * different
169     * permutable group).
170     * @return Segment The next child segment or null if there are no more
171     * children.
172     */
173     public Segment getNextChild() {
174         if ( childIterator == null ) throw new IllegalArgumentException( "Child
175             iterator has not been reset before use." );
176         if ( childIterator.hasNext() ) {
177             final Segment next = childIterator.next();
178             if ( permutationIterator != null ) {
179                 while ( permutationIterator.hasNext() ) {
180                     final Segment pnext = permutationIterator.next();

```

```

170         if ( pnext != this )
171             return pnext;
172     }
173     permutationIterator = null;
174 }
175 if ( next.getPermutation() != null ) {
176     permutationIterator =
177     next.getPermutation().getPermutation().iterator();
178     while ( permutationIterator.hasNext() ) {
179         final Segment pnext = permutationIterator.next();
180         if ( pnext != this )
181             return pnext;
182     }
183     throw new IllegalArgumentException( "Non-parent segment not found in
184     permutation." );
185 }
186 return next;
187 }
188
189 /*****
190  * Blocks
191  *****/
192
193 /**
194  * The block that the segment has been embedded in. This will be finalised
195  * once all segments have been embedded and blocks will not be concatenated.
196  */
197 private Block embeddedBlock = null;
198
199 /**
200  * Gets the blocks containing the child segments (and their descendants).
201  */
202 protected final LinkList<Block> childBlocks = new LinkList<Block>();
203
204 /**
205  * Returns the block the segment has been embedded in.
206  * @return Block The block that the segment has been embedded in.
207  */
208 public Block getEmbeddedBlock() { return embeddedBlock; }
209
210 /**
211  * Sets the block the segment has been embedded in.
212  * @param block
213  */
214 public void setEmbeddedBlock( final Block block ) {
215     if ( block == null ) throw new IllegalArgumentException( "Block argument
216     cannot be null." );
217     embeddedBlock = block;
218 }
219
220 /**
221  * Returns the last block in the list of blocks containing child segments.
222  * @return Block The last block in the list of blocks containing child
223  * segments.
224  */
225 public Block getLastChildBlock() { return childBlocks.getTail(); }
226
227 /**
228  * Adds the given block to the list of child blocks.
229  * @param block A non-null block
230  */
231 public void addChildBlock( final Block block ) { childBlocks.addTail( block
232 ); }

```



```

231  /**
232   * Removes the last block from the list of child blocks.
233   * @see Block#concatenate()
234   */
235  public void removeLastChildBlock() { childBlocks.removeTail(); }

238  /**
239   * Planarity Testing
240   * *****/
241  // public static final int NOT_EMBEDDABLE = 0;
242  public static final int CONFLICT_INSIDE = 0;
243  public static final int CONFLICT_BOTH_SIDES = 1;
244  public static final int EMBED_ON_INSIDE = 2;
245  public static final int FLIP_EMBED_ON_INSIDE = 3;
246  public static final int EMBED_ON_OUTSIDE = 4;
247  public static final int NEW_BLOCK = 5;

249  /**
250   *
251   */
252  private boolean isEmbedded = false;

254  /**
255   * The last segment that was embedded.
256   */
257  private Segment lastEmbeddedSegment = null;

259  /**
260   * Returns the last embedded segment.
261   * @return Segment
262   */
263  protected Segment getLastEmbeddedSegment() {
264      return lastEmbeddedSegment;
265  }

267  /**
268   * Checks whether the segment is embedded.
269   * @return boolean True if the segment has been embedded (or if no attempt
270   * has been
271   * made to embed the segment), false if the embedding has
272   * failed.
273   */
274  public boolean isEmbedded() { return isEmbedded; }

276  /**
277   * Sets whether the segment has been embedded.
278   * @param isEmbedded True if the segment has been embedded, false otherwise.
279   */
280  public void setIsEmbedded( final boolean isEmbedded ) {
281      this.isEmbedded = isEmbedded;
282  }

284  /**
285   * Sets the last embedded segment.
286   * @param segment
287   */
288  public void setLastEmbeddedSegment( final Segment segment ) {
289      lastEmbeddedSegment = segment;
290  }

291  /**
292   *
293   * @param segment
294   * @param block

```

```

292     * @return
293     */
294     public int checkEmbedding( Segment segment, Block block ) {
295         final boolean inConflicts = block.insideConflictsWith( segment );
296         final boolean outConflicts = block.outsideConflictsWith( segment );
297         if ( inConflicts ) {
298             final Segment inside = block.getLastInside();
299             inside.addConflictingSegment( segment );
300             segment.addConflictingSegment( inside );
301         }
302         if ( outConflicts ) {
303             final Segment outside = block.getLastOutside();
304             outside.addConflictingSegment( segment );
305             segment.addConflictingSegment( outside );
306         }
307         if ( Graph.DEBUG_EMBED ) System.out.println( "Embedding Segment: " +
            segment.toString() + " on " + toString() + " (" + inConflicts + " " +
            (block.isEmptyInside()?"null":block.getLastInside().toString()) + ", " +
            outConflicts + " " +
            (block.isEmptyOutside()?"null":block.getLastOutside().toString()) + ")" );
308         if ( inConflicts && outConflicts ) return CONFLICT_BOTH_SIDES;
309         if ( !inConflicts && !outConflicts ) return NEW_BLOCK;
310         if ( inConflicts ) {
311             if ( segment.isFlippable() ) return EMBED_ON_OUTSIDE;
312             else if ( block.isFlippable() ) return FLIP_EMBED_ON_INSIDE;
313             else return CONFLICT_INSIDE;
314         } else return EMBED_ON_INSIDE;
315     }
316
317     /**
318     * Tests whether a segment, that is a child of this segment, is embeddable.
319     * @param segment The non-null segment, that is a child of this segment,
320     * that is being embedded.
321     * @return True if the segment has been successfully embedded, false
322     * otherwise.
323     */
324     public boolean embedChildSegment( Segment segment ) {
325         if ( segment == null ) throw new IllegalArgumentException(
            "Cannot embed a null segment." );
326         if ( segment.getParent() != this ) throw new IllegalArgumentException(
            "Cannot embed a segment on a non-parent segment." );
327         Block childBlock = getLastChildBlock();
328
329         /*
330         * Fully embed all the segments in the last block that have a leaf
331         * segment with an equal or higher DFS index than the passed child
332         * segment's root vertex's DFS index. If all segments in the block
333         * are fully embedded then remove this block from the list (the
334         * block will remain in the graph's list of blocks) and repeat with
335         * the next latest block until either all blocks are processed or
336         * a block is found with segments that are not fully embedded.
337         */
338         while ( childBlock != null ) {
339             if ( !childBlock.embedTo( segment.getRootVertexDFSIndex() ) )
340                 break;
341             removeLastChildBlock();
342             childBlock = getLastChildBlock();
343         }
344
345         //
346         // Check for permutations with previously embedded segment.
347         //
348         if ( (segment.getLeafVertexDFSIndex() <
            segment.getRootVertexDFSIndex())
349             && (lastEmbeddedSegment != null)

```

```

350         && (lastEmbeddedSegment.getRootVertexDFSIndex() ==
351             segment.getRootVertexDFSIndex())
352         && (lastEmbeddedSegment.getLeafVertexDFSIndex() ==
353             segment.getLeafVertexDFSIndex())
354         && (lastEmbeddedSegment.getLowPoint2Index() >=
355             lastEmbeddedSegment.getRootVertexDFSIndex())
356         && (segment.getLowPoint2Index() >= segment.getRootVertexDFSIndex())
357     ) {
358         //
359         // If Child and lastEmbeddedChild have the same start and end points
360         // and
361         // do not have a low point 2 vertex on the parent (if they do have a
362         // low
363         // point 2 vertex on the parent they must be embedded adjacent to the
364         // parent and may be flippable but are not permutable).
365         // In this case the segments are permutable and the child must be
366         // embeddable
367         // within the previous segment (same start and end so do not overlap
368         // and
369         // neither connects to an ancestor).
370         //
371
372         if ( !lastEmbeddedSegment.isPermutable() ) {
373             lastEmbeddedSegment.permutation = new SegmentPermutation(
374                 lastEmbeddedSegment, segment );
375             getGraph().addPermutation( lastEmbeddedSegment.getPermutation() );
376         } else
377             lastEmbeddedSegment.getPermutation().addTail( segment );
378
379         segment.permutation = lastEmbeddedSegment.getPermutation();
380
381         if ( childBlock.getLastInside() == lastEmbeddedSegment )
382             childBlock.addSegmentInside( segment );
383         else if ( childBlock.getLastOutside() == lastEmbeddedSegment )
384             childBlock.addSegmentOutside( segment );
385         else
386             throw new IllegalArgumentException( "Permutable segment " +
387                 segment + " (" + segment.getRootVertexDFSIndex() + ", " +
388                 segment.getLeafVertexDFSIndex() + ", " +
389                 segment.getLowPoint2Index() + ") - cannot match last embedded
390                 segment " + lastEmbeddedSegment + " (" +
391                 lastEmbeddedSegment.getRootVertexDFSIndex() + ", " +
392                 lastEmbeddedSegment.getLeafVertexDFSIndex() + ", " +
393                 lastEmbeddedSegment.getLowPoint2Index() + ") to either side of
394                 last block. <" + childBlock.getLastInside() + ", " +
395                 childBlock.getLastOutside() + ">" );
396
397         lastEmbeddedSegment = segment;
398         segment.setIsEmbedded( true );
399         return true;
400     }
401
402     /*
403     * If the parent has no blocks (that are not already fully embedded)
404     * then the segment can be added without conflict.
405     */
406     if ( childBlock == null ) {
407         new Block( segment );
408         segment.setIsEmbedded( true );
409         return true;
410     }
411
412     /*
413     * Otherwise check the last block for conflicts.
414     */

```

```

397     final int status = checkEmbedding( segment, childBlock );
398     boolean embedSide = Block.INSIDE;
399     boolean embedSideEmpty = false;
400     switch ( status ) {
401     case CONFLICT_INSIDE:
402         segment.setIsEmbedded( false );
403         return false;
404     case CONFLICT_BOTH_SIDES:
405         segment.setIsEmbedded( false );
406         return false;
407     case NEW_BLOCK:
408         new Block( segment );
409         segment.setIsEmbedded( true );
410         return true;
411     case EMBED_ON_OUTSIDE:
412         embedSide = Block.OUTSIDE;
413         embedSideEmpty = childBlock.isEmptyOutside();
414         break;
415     case FLIP_EMBED_ON_INSIDE:
416         childBlock.flip();
417     case EMBED_ON_INSIDE:
418         embedSideEmpty = childBlock.isEmptyInside();
419     }
420     while ( embedSideEmpty && childBlock.getPrevious() != null ) {
421         final Block prevBlock = childBlock.getPrevious();
422         boolean prevEmbedSide = Block.INSIDE;

424         switch ( checkEmbedding( segment, prevBlock ) ) {
425         case CONFLICT_INSIDE:
426             segment.setIsEmbedded( false );
427             return false;
428         case CONFLICT_BOTH_SIDES:
429             segment.setIsEmbedded( false );
430             return false;
431         case NEW_BLOCK:
432             if ( embedSide == Block.INSIDE )
433                 childBlock.addSegmentInside( segment );
434             else
435                 childBlock.addSegmentOutside( segment );
436             segment.setIsEmbedded( true );
437             return true;
438         case FLIP_EMBED_ON_INSIDE:
439             prevBlock.flip();
440         case EMBED_ON_INSIDE:
441             prevEmbedSide = Block.INSIDE;
442             break;
443         case EMBED_ON_OUTSIDE:
444             prevEmbedSide = Block.OUTSIDE;
445         }
446         if ( prevEmbedSide != embedSide ) {
447             /*
448              * One block inside, one outside means that the segment
449              * must be flippable.
450              */
451             if ( prevBlock.isFlippable() ) {
452                 prevBlock.flip();
453             } else if ( childBlock.isFlippable() ) {
454                 childBlock.flip();
455                 embedSide = !embedSide;
456             } else {
457                 segment.setIsEmbedded( false );
458                 return false;
459             }
460         }
461         childBlock = childBlock.concatenate();

```

```

462     }
463     if ( embedSide == Block.INSIDE )
464         childBlock.addSegmentInside( segment );
465     else
466         childBlock.addSegmentOutside( segment );
467     segment.setIsEmbedded( true );
468     return true;
469 }

471 /**
472  * Appends the segments on the inside of all the non-flippable
473  * child blocks to the last child block of this segment's parent.
474  * <p>
475  * Flippable blocks are not appended as the segments within
476  * that block are all flippable and, hence, only connect to
477  * the root stem or leaf of their parent and do not connect
478  * to an ancestor. This means that they cannot conflict with
479  * a later embedded descendant of an ancestor.
480  */
481 public void appendDescendantSegments() {
482     if ( getParent() == null )
483         return;
484     final LinkedList<Segment> segments = new LinkedList<Segment>();
485     while( !childBlocks.isEmpty() ) {
486         Block childBlock = childBlocks.removeTail();
487         if ( !childBlock.isFlippable() )
488             segments.addAllHead( childBlock.getInsideSegments() );
489     }
490     getParent().getLastChildBlock().appendSegmentsAfter( this, segments );
491 }

493 /**
494  * Checks whether this segment conflicts with a previously embedded segment.
495  * @param segment A non-null segment to compare with this segment for a
496  * conflict.
497  * @return boolean
498  */
499 public boolean conflictsWith( Segment segment ) {
500     if ( segment == null ) throw new IllegalArgumentException( "Segment
501         argument cannot be null" );
502
503     final boolean conflict;
504     if ( this.getSegmentIndex() < segment.getSegmentIndex() ) {
505         final int leaf = this.getLeafVertexDFSIndex();
506         conflict = segment.getLeafVertexDFSIndex() < leaf && leaf <
507             segment.getRootVertexDFSIndex();
508     } else if ( this.getSegmentIndex() > segment.getSegmentIndex() ) {
509         final int leaf = segment.getLeafVertexDFSIndex();
510         conflict = this.getLeafVertexDFSIndex() < leaf && leaf <
511             this.getRootVertexDFSIndex();
512     } else
513         conflict = false;
514
515     // System.out.println(toPathString() + ".conflictsWith(" +
516     segment.toPathString() + ") = " + conflict);
517     return conflict;
518 }

519 /**
520  * Checks whether the segment can be embedded on either side of the parent.
521  * @return boolean True if the segment can be embedded on either side,
522  * false otherwise.
523  */
524 public boolean isFlippable() {
525     return parent != null && ( getLeafVertexDFSIndex() >=
526         parent.getRootVertexDFSIndex()

```

```

520         || ( getLeafVertexDFSIndex() ==
521             parent.getLeafVertexDFSIndex()
522             && getLowPoint2Index() >=
523                 parent.getRootVertexDFSIndex() ) );
524     }

525     /*****
526     * Ordering
527     *****/
528     /**
529     * The side of the parent segment on which this segment has been embedded.
530     */
531     private boolean embeddingSide = Block.INSIDE;

532     /**
533     * The direction that the segment is embedded.
534     */
535     private boolean embeddingDirection = CLOCKWISE;

536     /**
537     * The direction that the current child segment is being embedded.
538     */
539     private boolean childEmbeddingDirection = CLOCKWISE;

540     /**
541     * Sets the side of the parent segment on which this segment has been
542     * embedded.
543     * @param side Value matches either Block.INSIDE or Block.OUTSIDE.
544     */
545     public void setSide( final boolean side ) { embeddingSide =
546         side; }

547     /**
548     * Returns the side of the parent segment on which this segment has been
549     * embedded.
550     * @return boolean Returned value matches either Block.INSIDE or
551     * Block.OUTSIDE.
552     */
553     public boolean getSide() {
554         if ( getEmbeddedBlock() == null || !getEmbeddedBlock().isInverted() )
555             return embeddingSide;
556         return !embeddingSide;
557     }

558     /**
559     * Sets the direction that the segment is embedded.
560     * @param direction
561     */
562     public void setEmbeddedDirection( final boolean direction ) {
563         embeddingDirection = direction; }

564     /**
565     * Gets the direction the segment is embedded.
566     * @return boolean Returned value matches either Segment.CLOCKWISE or
567     * Segment.ANTICLOCKWISE.
568     */
569     public boolean getEmbeddingDirection() { return
570         embeddingDirection; }

571     /**
572     * Sets the direction that the segment is embedded.
573     * @param direction
574     */

```

```

577     public void setChildEmbeddedDirection( final boolean direction ) {
578         childEmbeddingDirection = direction;
579         if ( getParent() != null ) {
580             root.setChildEmbeddingDirection( stem.getHead(), direction );
581             if ( stem.getTail().isDFSFrond() )
582                 leaf.setChildEmbeddingDirection( stem.getTail(), direction );
583         }
584     }

586     /**
587     * Gets the direction the current child of the segment is being embedded.
588     * @return boolean Returned value matches either Segment.CLOCKWISE or
589     * Segment.ANTICLOCKWISE.
590     */
591     public boolean getChildEmbeddingDirection() { return
childEmbeddingDirection; }

592     /**
593     * Permutations
594     */

596     private SegmentPermutation permutation = null;

598     public SegmentPermutation getPermutation() { return permutation; }

600     public void setPermutation( final SegmentPermutation p ) {
601         this.permutation = p;
602     }

604     public boolean isPermutable() { return permutation != null; }

606     public boolean isPermutableWithParent() {
607         return isPermutable() &&
608             getEmbeddedBlock() != null &&
609             getEmbeddedBlock().isFlippable() &&
610             getEmbeddedBlock().isEmptyOutside();
611     }

613     /**
614     * Non-planarity
615     */

617     /**
618     * The segments that conflict with this segment during embedding.
619     */
620     private final UnmodifiableLinkedList<Segment> conflictingSegments = new
UnmodifiableLinkedList<Segment>();

622     /**
623     * Add a conflict with another segment.
624     * @param segment The non-null conflicting segment
625     */
626     public void addConflictingSegment( final Segment segment ) {
627         if ( segment == null ) throw new IllegalArgumentException( "Cannot
conflict with a null segment." );
628         conflictingSegments.addTail( segment );
629     }

631     /**
632     * Get the list of conflicting segments.
633     * @return
634     */
635     public UnmodifiableLinkedList<Segment> getConflictingSegments() { return
conflictingSegments; }

```

```

637  /*****
638  * Statistics
639  *****/
640  private int depth = 0;

642  /**
643  * Gets the depth of the segment within the segment tree.
644  * @return int The depth of the segment within the segment tree.
645  */
646  public int getDepth() { return depth; }

648  /**
649  * Sets the depth equal to the parent segment's depth incremented by one.
650  */
651  public void setDepthFromParent() {
652      if ( getParent() == null )
653          depth = 0;
654      else
655          depth = getParent().getDepth() + 1;
656  }

658  /*****
659  * String Output
660  *****/
661  /**
662  * Returns a string representation of the root, stem and leaf
663  * vertices.
664  *
665  * @return String
666  */
667  public String toString() {
668      StringBuffer b = new StringBuffer();
669      toBufferedString( b );
670      return b.toString();
671  }

673  private void toBufferedString( StringBuffer b ) {
674      b.append( '[' );
675      b.append( root );
676      for ( final Edge e: stem ) {
677          b.append( ", " );
678          b.append( e.getDFSTo() );
679      }
680      b.append( ']' );
681  }

683  private void toBufferedIndentedString( StringBuffer buffer, String indent,
684  int d, String indentIncrement ) {
685      buffer.append( indent );
686      for ( int i = 0; i < d; i++ )
687          buffer.append( indentIncrement );
688      toBufferedString( buffer );
689      buffer.append( " (" );
690      buffer.append( getSide() == Block.INSIDE? 'I': 'O' );
691      buffer.append( ")\n" );
692      for ( Segment child: childSegments )
693          child.toBufferedIndentedString( buffer, indent, d+1, indentIncrement );
694  }

696  public String toIndentedString( String indent, String indentIncrement ) {
697      StringBuffer buffer = new StringBuffer();
698      toBufferedIndentedString( buffer, indent, 0, indentIncrement );
699      return buffer.toString();
700  }

```



```

701 public int toLaTeXString( StringBuffer buffer ,
702                          java.util.HashMap<Segment, java.lang.Character>
703                          segmentMap,
704                          int indent,
705                          int line ,
706                          boolean showSide ) {
707     buffer.append( "\t\t\t\node (" );
708     buffer.append( segmentMap.get( this ) );
709     buffer.append( ") at (" );
710     buffer.append( indent );
711     buffer.append( "\\unit,-" );
712     buffer.append( line );
713     buffer.append( "\\unit) [label={[name=" );
714     buffer.append( segmentMap.get( this ) );
715     buffer.append( "',style=noderect]right:\\SquashFont{ }" );
716     toBufferedString( buffer );
717     if (showSide) {
718         buffer.append( " (" );
719         buffer.append( getSide() == Block.INSIDE? 'I': 'O' );
720         buffer.append( ")\n" );
721     }
722     buffer.append( "}] {};\n" );
723     if ( this.getParent() != null ) {
724         buffer.append( "\t\t\t\tdraw (node cs:name=" );
725         buffer.append( segmentMap.get( this.getParent() ) );
726         buffer.append( "',anchor=south west) |- (" );
727         buffer.append( segmentMap.get( this ) );
728         buffer.append( ");\n" );
729     }
730     line = line + 1;
731     for ( Segment child: childSegments )
732         line = child.toLaTeXString( buffer, segmentMap, indent + 1, line,
733                                     showSide );
734     return line;
735 }

```

D.2.10 mgtaylor.graph.SegmentPermutation

```

1 package mgtaylor.graph;
2
3 import mgtaylor.datastructures.MutableUnmodifiableLinkedList;
4
5 public class SegmentPermutation extends MutableUnmodifiableLinkedList<Segment> {
6     private static final boolean DEBUG = false;
7
8     private final Segment parentSegment;
9     private boolean isPermutableAboutParent = false;
10
11     public SegmentPermutation( final Segment... array ) {
12         if ( array.length == 0 ) throw new IllegalArgumentException( "Cannot
13             initialise a segment permutation with no segments." );
14         for ( Segment element: array )
15             addTail( element );
16         parentSegment = array[0].getParent();
17         if ( DEBUG ) System.out.println( "SegmentPermutation(): " + parentSegment
18             + ", " + this );
19     }
20
21     public Segment getParentSegment() { return parentSegment; }
22     public int getTotalPermutations() { return constraints.size() ==
23         1?super.getTotalPermutations()/2:super.getTotalPermutations(); }

```

```

22  @Override
23  public void handleSwap( final Segment earlier , final Segment later ) {
24      if ( earlier == parentSegment )
25          later.setSide( Block.INSIDE );
26      else if ( later == parentSegment )
27          earlier.setSide( Block.OUTSIDE );
28  }

30  @Override
31  public void nextPermutation() {
32      if ( constraints.size() == 1 &&
33          getCurrentPermutation() >= super.getTotalPermutations()/2 - 1 )
34          jumpToPermutation( 0 );
35      else
36          super.nextPermutation();
37  }

39  @Override
40  public void previousPermutation() {
41      if ( constraints.size() == 1 &&
42          getCurrentPermutation() <= 0 )
43          jumpToPermutation( super.getTotalPermutations()/2 - 1 );
44      else
45          super.previousPermutation();
46  }

48  @Override
49  public void jumpToPermutation( final int p ) {
50      super.jumpToPermutation( p );
51      orderPermutationsWithParent();
52  }

54  public boolean isPermutableWithParent() {
55      return isPermutableAboutParent;
56  }

58  public void setIsPermutableAboutParent() {
59      if ( DEBUG ) System.out.println(
60          "SegmentPermutation.setIsPermutableAboutParent(): " + this + ", " +
61          parentSegment + " (" + isPermutableAboutParent + ")");
62      if ( parentSegment != null && !isPermutableAboutParent ) {
63          Segment segment = getHead();
64          if ( segment.isPermutableWithParent() ) {
65              isPermutableAboutParent = true;
66              addHead( parentSegment );
67          } else if ( parentSegment.getParent() == null &&
68              parentSegment.getChildSegments().getHead() == segment &&
69              segment.getEmbeddedBlock().hasNoSegmentsOutside() ) {
70              addHead( parentSegment );
71              isPermutableAboutParent = true;
72              addConstraint( parentSegment, segment );
73          } else
74              isPermutableAboutParent = false;
75      }
76  }

76  public void orderPermutationsWithParent() {
77      if ( parentSegment == null )
78          return;
79      boolean beforeParent = true;
80      for ( Segment segment: getPermutation() ) {
81          if ( segment == parentSegment )
82              beforeParent = false;
83          else if ( beforeParent ) {

```

```

84         segment.setSide( Block.OUTSIDE );
85     } else {
86         segment.setSide( Block.INSIDE );
87     }
88 }
89 }
90 }

```

D.2.11 mgtaylor.graph.Vertex

```

1 package mgtaylor.graph;
2
3 import java.util.ArrayList;
4
5 import mgtaylor.datastructures.UnmodifiableLinkedList;
6 import mgtaylor.datastructures.LinkList;
7
8 public class Vertex implements GraphObject {
9     /**
10      * The graph the vertex belongs to.
11      */
12     private final Graph graph;
13
14     /**
15      * The vertex's index within the graph's array of vertices.
16      * graph.getVertices().get( this.index ) == this
17      */
18     private final int index;
19
20     /**
21      * The maximum number of edges that can be connected to the vertex.
22      */
23     private final int capacity;
24
25     /**
26      * The id of the vertex.
27      */
28     private final String id;
29
30     /**
31      * The array of edges connected to the vertex.
32      */
33     private final ArrayList<Edge> connectedEdges;
34
35     /**
36      * Creates a vertex in the graph.
37      * @param graph The graph the vertex is part of.
38      * @param index The index of the vertex within the graph's array of
39      * vertices.
40      * @param id The label for the vertex.
41      * @param capacity The number of edges the vertex is assumed to be
42      * connected to.
43      */
44     public Vertex( final Graph graph, final int index, final String id, final int
45     capacity ) {
46         if ( graph == null ) throw new IllegalArgumentException( "Graph argument
47         cannot be null." );
48         if ( id == null ) throw new IllegalArgumentException( "Id argument
49         cannot be null." );
50         if ( capacity < 0 ) throw new IllegalArgumentException( "Capacity
51         argument cannot be negative." );
52         this.graph = graph;

```

```

47     this.index = index;
48     this.id    = id;
49     this.capacity = capacity;
50     connectedEdges = new ArrayList<Edge>( capacity );
51 }
52
53 /**
54  * The graph containing the vertex.
55  * @return Graph
56  */
57 public Graph getGraph()                { return graph; }
58
59 /**
60  * Gets the index of the vertex within the graph's array of vertices.
61  * @return
62  */
63 public int getIndex()                  { return index; }
64
65 /**
66  * The string representation of the vertex.
67  */
68 public String toString()               { return id; }
69
70 /**
71  * The number of edges connected to the vertex.
72  * @return int
73  */
74 public int size()                      { return connectedEdges.size(); }
75
76 /**
77  * Returns the edges connected to the vertex.
78  * @return ArrayList<Edge>
79  */
80 public ArrayList<Edge> getConnectedEdges() { return connectedEdges; }
81
82 /**
83  * Connects the given edge to the vertex.
84  * @param edge - A non-null edge that contains this vertex.
85  * @return int - The index within the Vertex's array of edges
86  *               corresponding to the location of the edge.
87  */
88 public int connectTo( final Edge edge ) {
89     if ( edge == null )                 throw new IllegalArgumentException(
90         "Edge cannot be null." );
91     if ( connectedEdges.size() >= capacity ) throw new
92         IllegalArgumentException( "Vertex has reached capacity: " + toString() +
93         ", " + edge.toString() );
94     if ( !edge.contains( this ) )       throw new IllegalArgumentException(
95         "Edge does not connect to this vertex." );
96     connectedEdges.add( edge );
97     return connectedEdges.size() - 1;
98 }
99
100 /**
101  * Checks to see if the edge contains this vertex.
102  * @param Edge - A non-null edge
103  * @return boolean
104  */
105 @Override
106 public boolean isConnectedTo( final Edge edge ) {
107     if ( edge == null )                 throw new IllegalArgumentException( "Edge cannot
108         be null." );
109     return edge.contains( this );
110 }

```

```

107  /*****
108  * DFS
109  *****/
110  public static final int DFS_UNSET = -1;

112  /**
113   * Depth First Search Index for the vertex.
114   * This is a sequentially incremented identifier
115   */
116  private int dfsIndex = DFS_UNSET;

118  /**
119   * DFS Index for the lowest possible vertex reachable by
120   * traversing descendants and back edges of the DFS tree.
121   */
122  private int lowPoint1 = DFS_UNSET;

124  /**
125   * DFS Index for the second lowest possible vertex reachable
126   * by traversing descendants and back edges of the DFS tree.
127   */
128  private int lowPoint2 = DFS_UNSET;

130  /**
131   * Edge connecting the vertex to its parent in the DFS tree.
132   */
133  private Edge dfsParent = null;

135  /**
136   * The edges in the DFS tree that are outbound from the current
137   * edge.
138   */
139  private UnmodifiableLinkedList<Edge> dfsAdjacency = null;

141  /**
142   * Resets the DFS index of the vertex.
143   */
144  public void resetDFSIndex() {
145      dfsIndex = DFS_UNSET;
146  }

148  /**
149   * Returns the DFS index of the vertex.
150   * @return int
151   */
152  public int getDFSIndex() { return dfsIndex; }

154  /**
155   * Compares to see if this vertex has a lower DFS index than the given vertex.
156   * @param vertex
157   * @return
158   */
159  public boolean lessThan( final Vertex vertex ) {
160      return this.getDFSIndex() < vertex.getDFSIndex();
161  }

163  /**
164   * Compares to see if this vertex has a greater DFS index than the given
165   * vertex.
166   * @param vertex
167   * @return
168   */
169  public boolean greaterThan( final Vertex vertex ) {
170      return this.getDFSIndex() > vertex.getDFSIndex();

```

```

172  /**
173   * Compares to see if this vertex has a lower or equal DFS index than the
      given vertex.
174   * @param vertex
175   * @return
176   */
177  public boolean lessThanOrEqualTo( final Vertex vertex ) {
178      return this.getDFSIndex() <= vertex.getDFSIndex();
179  }

181  /**
182   * Compares to see if this vertex has a greater or equal DFS index than the
      given vertex.
183   * @param vertex
184   * @return
185   */
186  public boolean greaterThanOrEqualTo( final Vertex vertex ) {
187      return this.getDFSIndex() >= vertex.getDFSIndex();
188  }

190  /**
191   * Compares the vertices to find the one with the lowest DFS index.
192   * @param vertices
193   * @return
194   */
195  public static Vertex min( final Vertex... vertices ) {
196      if ( vertices.length == 0 ) throw new IllegalArgumentException( "Zero
      vertices." );
197      Vertex min = vertices[0];
198      for ( int i = 1; i < vertices.length; i++ )
199          if ( vertices[i].lessThan( min ) )
200              min = vertices[i];
201      return min;
202  }

204  /**
205   * Compares the vertices to find the one with the highest DFS index.
206   * @param vertices
207   * @return Vertex
208   */
209  public static Vertex max( final Vertex... vertices ) {
210      if ( vertices.length == 0 ) throw new IllegalArgumentException( "Zero
      vertices." );
211      Vertex max = vertices[0];
212      for ( int i = 1; i < vertices.length; i++ )
213          if ( vertices[i].greaterThan( max ) )
214              max = vertices[i];
215      return max;
216  }

218  /**
219   * Returns the first DFS low point.
220   * @return int
221   */
222  public int getLowPoint1() { return lowPoint1; }

224  /**
225   * Returns the second DFS low point.
226   * @return int
227   */
228  public int getLowPoint2() { return lowPoint2; }

230  /**
231   * Sets the first DFS low point for the vertex.

```

```

232     * @param int index
233     */
234     public void setLowPoint1( final int index ) { lowPoint1 = index; }

236     /**
237     * Sets the second DFS low point for the vertex.
238     * @param int index
239     */
240     public void setLowPoint2( final int index ) { lowPoint2 = index; }

242     /**
243     * Returns the edge to the parent vertex in the DFS tree.
244     * @return Edge
245     */
246     public Edge getDFSParent() { return dfsParent; }

248     /**
249     * Sets the Edge linking the vertex to its parent in the
250     * DFS tree.
251     * @param Edge edge
252     */
253     public void setDFSParent( final Edge edge ) {
254         dfsParent = edge;
255         dfsIndex = lowPoint1 = lowPoint2 = graph.getDFSIndex();
256         graph.incrementDFSIndex();
257         dfsAdjacency = new UnmodifiableLinkedList<Edge>();
258     }

260     /**
261     * Adds an edge to the vertex, such that it is an outbound uni-directional
262     * edge used
263     * to create the DFS palm tree.
264     * @param edge - Non-null edge
265     */
266     public void addDFSAdjacentEdge( final Edge edge ) {
267         if ( edge == null )
268             throw new IllegalArgumentException( "Edge cannot be null." );
269         dfsAdjacency.addTail( edge );
270     }

272     /**
273     * Gets the list of outbound edges used to create the DFS palm tree.
274     *
275     * @return {@linkplain UnmodifiableLinkedList}&lt;Edge>
276     */
277     public UnmodifiableLinkedList<Edge> getDFSAdjacency() {
278         return dfsAdjacency;
279     }

281     /**
282     * Segment
283     * *****/
284     private Segment segment = null;

286     /**
287     * Returns the segment where this vertex is part of the stem.
288     * @return Segment
289     */
290     public Segment getSegment() { return segment; }

292     /**
293     * Sets the segment where this vertex is part of the stem.
294     * @param segment
295     */

```

```

296 public void setSegment( final Segment segment ) { this.segment = segment; }

298 /*****
299  * Order
300  *****/

302 /**
303  * <p>The first inbound DFS edge; typically this links the vertex to it's
304  * parent
305  * in the DFS tree but in the case of the root vertex, which has no parent,
306  * it is the frond edge terminating the first (and cyclic) segment. This edge
307  * is
308  * used as the first reference point to order the other edges connected to
309  * this
310  * vertex.<p>
311  */
312 private Edge orderParentEdge = null;

313 /**
314  * The first outbound edge from the vertex within the DFS tree.
315  */
316 private Edge orderStemEdge = null;

317 /**
318  * Returns the edge that links the vertex to it's parent in the DFS tree; in
319  * the
320  * case of the root vertex, which does not have a parent vertex, this is the
321  * frond
322  * edge terminating the initial (cyclic) segment.
323  * @return Edge
324  */
325 public Edge getEdgeOrderParent() { return orderParentEdge; }

326 /**
327  * Sets the edge that links the vertex to it's parent in the DFS tree; in the
328  * case of the root vertex, which does not have a parent vertex, this is the
329  * frond
330  * edge terminating the initial (cyclic) segment.
331  * @param edge
332  */
333 public void setEdgeOrderParent( final Edge edge ) {
334     if ( edge == null ) throw new IllegalArgumentException( "Cannot
335     order a null edge." );
336     if ( !edge.contains( this ) ) throw new IllegalArgumentException( "Edge
337     does not connect to this vertex." );
338     orderParentEdge = orderStemEdge = edge;
339 }

340 /**
341  * Returns the edge that is the first outbound edge from the vertex in the
342  * DFS tree
343  * (and the vertex's outbound edge contained in this vertex's segment).
344  * @return
345  */
346 public Edge getEdgeOrderStem() { return orderStemEdge; }

347 /**
348  * Returns the edge that is the first outbound edge from the vertex in the
349  * DFS tree
350  * (and the vertex's outbound edge contained in this vertex's segment).
351  * @return
352  */
353 public void setEdgeOrderStem( final Edge edge ) {

```



```

350         if ( edge == null )                throw new IllegalArgumentException( "Cannot
order a null edge." );
351         if ( !edge.contains( this ) ) throw new IllegalArgumentException( "Edge
does not connect to this vertex." );
352         if ( orderParentEdge == null ) {
353             orderParentEdge = orderStemEdge = edge;
354         } else {
355             orderStemEdge = edge;
356             orderParentEdge.setOrderAntiClockwise( this, edge );
357         }
358     }

359
360     /**
361     * Returns the edge that is ordered clockwise from the given edge
362     * connected to this vertex.
363     * @param edge An edge that is connected to this vertex.
364     * @return Edge
365     */
366     public Edge getClockwiseFrom( final Edge edge ) {
367         return edge.getOrderClockwiseFrom( this );
368     }

369
370     /**
371     * Returns the edge that is ordered anti-clockwise from the given edge
372     * connected to this vertex.
373     * @param edge An edge that is connected to this vertex.
374     * @return Edge
375     */
376     public Edge getAntiClockwiseFrom( final Edge edge ) {
377         return edge.getOrderAntiClockwiseFrom( this );
378     }

379
380     /**
381     * Orders the ordering edge so that it is clockwise from the reference edge.
382     * @param referenceEdge
383     * @param orderingEdge
384     */
385     public void connectClockwiseFrom( final Edge referenceEdge, final Edge
orderingEdge ) {
386         if ( orderingEdge == null )                throw new
IllegalArgumentException( "Edge being ordered cannot be null." );
387         if ( !orderingEdge.contains( this ) )      throw new
IllegalArgumentException( "Edge " + toString() + ":" + orderingEdge + "
being ordered does not connect to this vertex." );
388         if ( orderingEdge.isOrderedAbout( this ) ) throw new
IllegalArgumentException( "Edge " + toString() + ":" + orderingEdge + "
being ordered has already been ordered. " + referenceEdge );
389         if ( referenceEdge == null )              throw new
IllegalArgumentException( "Edge used to determine order position cannot be
null." );
390         if ( !referenceEdge.contains( this ) )      throw new
IllegalArgumentException( "Edge " + toString() + ":" + referenceEdge + "
used to determine order position does not connect to this vertex." );
391         if ( !referenceEdge.isOrderedAbout( this ) ) throw new
IllegalArgumentException( "Edge " + toString() + ":" + referenceEdge + "
used to determine order position has not been ordered. " + orderingEdge );
392         if ( orderParentEdge == null )            throw new
IllegalArgumentException( "Cannot order the given edge until both parent
and stem edge have been set." );
393         if ( orderStemEdge == null )              throw new
IllegalArgumentException( "Cannot order the given edge until the stem edge
has been set." );
394         referenceEdge.setOrderClockwise( this, orderingEdge );
395     }

```

```

397  /**
398  * Orders the ordering edge so that it is anti-clockwise from the reference
399  * edge.
400  * @param referenceEdge
401  * @param orderingEdge
402  */
403  public void connectAntiClockwiseFrom( final Edge referenceEdge, final Edge
orderingEdge ) {
404      if ( orderingEdge == null )                throw new
IllegalArgumentException( "Edge being ordered cannot be null." );
405      if ( !orderingEdge.contains( this ) )      throw new
IllegalArgumentException( "Edge " + toString() + ":" + orderingEdge + "
being ordered does not connect to this vertex." );
406      if ( orderingEdge.isOrderedAbout( this ) ) throw new
IllegalArgumentException( "Edge " + toString() + ":" + orderingEdge + "
being ordered has already been ordered. " + referenceEdge );
407      if ( referenceEdge == null )              throw new
IllegalArgumentException( "Edge used to determine order position cannot be
null." );
408      if ( !referenceEdge.contains( this ) )    throw new
IllegalArgumentException( "Edge " + toString() + ":" + referenceEdge + "
used to determine order position does not connect to this vertex." );
409      if ( !referenceEdge.isOrderedAbout( this ) ) throw new
IllegalArgumentException( "Edge " + toString() + ":" + referenceEdge + "
used to determine order position has not been ordered. " + orderingEdge );
410      if ( orderParentEdge == null )            throw new
IllegalArgumentException( "Cannot order the given edge until both parent
and stem edge have been set." );
411      if ( orderStemEdge == null )              throw new
IllegalArgumentException( "Cannot order the given edge until the stem edge
has been set." );
412      referenceEdge.setOrderAntiClockwise( this, orderingEdge );
413  }

414  public LinkedList<Edge> getCurrentlyOrderedEdges() {
415      final LinkedList<Edge> orderedEdges = new LinkedList<Edge>();
416      if ( orderParentEdge != null ) {
417          orderedEdges.addTail( orderParentEdge );
418          Edge edge = orderParentEdge.getOrderClockwiseFrom( this );
419          while ( edge != orderParentEdge ) {
420              orderedEdges.addTail( edge );
421              edge = edge.getOrderClockwiseFrom( this );
422          }
423      }
424      return orderedEdges;
425  }

426  /**
427  *
428  * @return
429  */
430  public String toOrderedEdgesString() {
431      StringBuffer b = new StringBuffer();
432      b.append( '<' );
433      boolean first = true;
434      for ( final Edge edge: getCurrentlyOrderedEdges() ) {
435          if ( first )
436              first = false;
437          else
438              b.append( ", " );
439          b.append( edge.getOtherEndFrom( this ).toString() );
440      }
441      b.append( '>' );
442      return b.toString();
443  }
444  }

```

```

446  /**
447   * Returns the clockwise ordering of edges connected to this vertex, starting
448   * from the parent edge.
449   * @return Edge[] – The array of ordered edges.
450   * @throws IllegalArgumentException If the parent edge is null or if the
451   * edges connected
452   * to the vertex have not all been ordered.
453   */
454  public Edge[] getEdgeOrder() {
455      if ( orderParentEdge == null )    throw new IllegalArgumentException(
456          "Vertex has not been ordered." );
457      final LinkedList<Edge> orderedEdges = getCurrentlyOrderedEdges();
458      if ( orderedEdges.size() != size() )    throw new
459          IllegalArgumentException( "Vertex (" + toString() + ") not fully ordered:
460          " + orderedEdges );
461      final Edge[] ordered = new Edge[ size() ];
462      int i = 0;
463      for ( final Edge e: orderedEdges )
464          ordered[i++] = e;
465      return ordered;
466  }

467  public void orderOutboundEdge( final Edge edge ) {
468      if ( edge == null )    throw new IllegalArgumentException(
469          "Outbound Edge cannot be null." );
470      if ( !edge.contains( this ) )    throw new IllegalArgumentException(
471          "Outbound Edge does not connect to this vertex." );
472
473      /*
474       * Line below is valid except in the case where the parent's frond is inbound
475       * to a vertex on parent's stem.
476       * if ( !orderEdgeStack.isEmpty() ) throw new IllegalArgumentException( "Edge
477       * Stack is not empty: " + edge + " – " + orderEdgeStack.toString());
478       */

479      if ( getSegment().getChildEmbeddingDirection() == Segment.CLOCKWISE )
480          this.connectAntiClockwiseFrom( orderParentEdge, edge );
481      else
482          this.connectClockwiseFrom( orderParentEdge, edge );
483      orderEdgeStack.addTail( new OrderingEdgeBoundaries( edge ) );
484      if ( Graph.DEBUG_ORDER ) System.out.println( toString() + ": Order
485      Outbound " + edge + " - " + (getSegment().getChildEmbeddingDirection() ==
486      Segment.CLOCKWISE?"AC":"C") + " from " + orderParentEdge + " - " +
487      toOrderedEdgesString() );
488  }

489  public void orderInboundEdge( final Edge edge ) {
490      if ( orderEdgeStack.isEmpty() ) {
491          if ( getSegment().getChildEmbeddingDirection() == Segment.CLOCKWISE )
492              this.connectClockwiseFrom( orderStemEdge, edge );
493          else
494              this.connectAntiClockwiseFrom( orderStemEdge, edge );
495          if ( Graph.DEBUG_ORDER ) System.out.println( toString() + ": Order
496          Inbound " + edge + " - " + (getSegment().getChildEmbeddingDirection() ==
497          Segment.CLOCKWISE?"C":"AC") + " from " + orderStemEdge + " - " +
498          toOrderedEdgesString() );
499      } else {
500          final OrderingEdgeBoundaries last = orderEdgeStack.getTail();
501          final boolean direction = last.childEmbeddingDirection;
502          if ( orderEdgeStack.size() == 1 && last.edge.getDFSFrom() == this ) {
503              // Stack only has outbound edge on it.
504              if ( direction == Segment.CLOCKWISE )
505                  this.connectClockwiseFrom( last.edge, edge );
506              else

```

```

496         this.connectAntiClockwiseFrom( last.edge, edge );
497         if ( Graph.DEBUG_ORDER ) System.out.println( toString() + ": Order
Inbound " + edge + " - " + (last.childEmbeddingDirection ==
Segment.CLOCKWISE?"C":"AC") + " from " + last.edge + " - " +
toOrderedEdgesString() );
498     } else {
499         // Latest on stack is an inbound edge.
500         if ( direction == Segment.CLOCKWISE ) {
501             this.connectClockwiseFrom( last.antiClockwiseBoundary, edge );
502             if ( Graph.DEBUG_ORDER ) System.out.println( toString() + ":
Order Inbound " + edge + " - C from " +
last.antiClockwiseBoundary + " - " + toOrderedEdgesString() );
503         } else {
504             this.connectAntiClockwiseFrom( last.clockwiseBoundary, edge );
505             if ( Graph.DEBUG_ORDER ) System.out.println( toString() + ":
Order Inbound " + edge + " - AC from " + last.clockwiseBoundary +
" - " + toOrderedEdgesString() );
506         }
507     }
508 }
509 orderEdgeStack.addTail( new OrderingEdgeBoundaries( edge ) );
510 }

512 public void orderRemoveEdge( final Edge edge ) {
513     if ( edge == null ) throw new IllegalArgumentException( "Edge
argument cannot be null." );
514     if ( orderEdgeStack.isEmpty() ) throw new IllegalArgumentException( "Edge
stack is empty: " + edge );
515     final OrderingEdgeBoundaries last = orderEdgeStack.removeTail();
516     if ( last.edge != edge ) throw new IllegalArgumentException( "Edge does
not match last edge on stack: " + toString() + ", " + orderEdgeStack + "
[" + last + ", " + edge + "]" );
517 }

519 public void setChildEmbeddingDirection( final Edge edge, final boolean
direction ) {
520     if ( edge == null ) throw new IllegalArgumentException( "Edge
argument cannot be null." );
521     if ( orderEdgeStack.isEmpty() ) throw new IllegalArgumentException( "Edge
stack is empty. " + edge.toString() );
522     final OrderingEdgeBoundaries last = orderEdgeStack.getTail();
523     /*
524      * Deal with case when a cycle is added from an articulation vertex.
525      * The outbound edge is added then the inbound edge is added to the same
526      * vertex so, instead of matching the last edge, the penultimate edge
527      * should match.
528      */
529     if ( last.edge != edge ) {
530         final OrderingEdgeBoundaries penul =
orderEdgeStack.getPenultimateTail();
531         if ( penul == null )
532             throw new IllegalArgumentException( "Edge " + edge + " does not
match last, and only, edge on stack " + orderEdgeStack );
533         if ( penul.edge != edge )
534             throw new IllegalArgumentException( "Edge " + edge + " does not
match last or penultimate edge on stack " + orderEdgeStack );
535         if ( penul.edge.getDFSFrom() != last.edge.getDFSFrom() )
536             throw new IllegalArgumentException( "Edge " + edge + " - mismatch
between penultimate and last edges on stack " + orderEdgeStack );
537         penul.childEmbeddingDirection = direction;
538     }
539     last.childEmbeddingDirection = direction;
540 }

```

```

542 private LinkedList<OrderingEdgeBoundaries> orderEdgeStack = new
    LinkedList<OrderingEdgeBoundaries>();

544 private class OrderingEdgeBoundaries {
545     private final Edge edge;
546     private final Edge clockwiseBoundary;
547     private final Edge antiClockwiseBoundary;
548     private boolean childEmbeddingDirection = Segment.CLOCKWISE;

550     private OrderingEdgeBoundaries( final Edge edge ) {
551         this.edge = edge;
552         clockwiseBoundary = Vertex.this.getClockwiseFrom( edge );
553         antiClockwiseBoundary = Vertex.this.getAntiClockwiseFrom( edge );
554     }

556     public String toString() {
557         return "[" + clockwiseBoundary + ", " + edge + ", " +
            antiClockwiseBoundary + "]";
558     }
559 }

561 }

```

D.3 Graph Type Files

D.3.1 mgtaylor.graph.graphTypes.OrderedFaceTrisectionGraph

```

1 package mgtaylor.graph.graphTypes;

3 import mgtaylor.graph.Graph;
4 import mgtaylor.graph.Vertex;
5 import mgtaylor.datastructures.LinkList;
6 import mgtaylor.datastructures.UnmodifiableLinkList;

8 import java.lang.Math;
9 import java.util.ArrayList;

11 public class OrderedFaceTrisectionGraph extends Graph {

13     public OrderedFaceTrisectionGraph(int numberOfVertices) {
14         super(numberOfVertices,
15             Math.max(3*numberOfVertices-6, Math.max(numberOfVertices-1, 0)));
16         switch (numberOfVertices) {
17             case 1:
18                 createVertex( 0, "1", 0);
19                 break;
20             case 2:
21                 createEdge( createVertex( 0, "1", 1 ), createVertex( 1, "2", 1 ) );
22                 break;
23             default:
24                 final VertexHolder[] v = new VertexHolder[ numberOfVertices ];
25                 final ArrayList<EdgeHolder> e = new ArrayList<EdgeHolder>( 3*
26                     numberOfVertices - 6 );
27                 final LinkList<FaceHolder> f = new LinkList<FaceHolder>();
28                 for ( int i = 0; i < numberOfVertices; i++ )
29                     v[i] = new VertexHolder( i );
30                 f.addTail( new FaceHolder( v[0], v[1], v[2] ) );
31                 e.add( new EdgeHolder( v[0], v[1] ) );
32                 e.add( new EdgeHolder( v[0], v[2] ) );
33                 e.add( new EdgeHolder( v[1], v[2] ) );

```

```

32         for ( int i = 3; i < numberOfVertices; i++ )
33             f.addAllTail( f.removeHead().splitFace( v[i], e));
34         for ( int i = 0; i < numberOfVertices; i++ )
35             v[i].generateVertex();
36         for ( int i = 0; i < 3*numberOfVertices-6; i++ )
37             e.get(i).generateEdge();
38     }
39 }

41 protected class VertexHolder {
42     private final int index;
43     private final UnmodifiableLinkedList<EdgeHolder> edges = new
UnmodifiableLinkedList<EdgeHolder>();

45     protected VertexHolder( final int i ) {
46         index = i;
47     }

49     protected void connectEdge( final EdgeHolder edge ) {
50         edges.addTail( edge );
51     }

53     protected void generateVertex() {
54         createVertex( index, Integer.toString( index + 1 ), edges.size() );
55     }

57     protected int getIndex() { return index; }
58 }

60 protected class EdgeHolder {
61     private final VertexHolder from;
62     private final VertexHolder to;

64     protected EdgeHolder( final VertexHolder f, final VertexHolder t ) {
65         from = f;
66         to = t;
67         from.connectEdge( this );
68         to.connectEdge( this );
69     }

71     protected void generateEdge() {
72         ArrayList<Vertex> vertices = getVertices();
73         createEdge( vertices.get( from.getIndex() ), vertices.get(
to.getIndex() ) );
74     }
75 }

77 protected class FaceHolder {
78     private final VertexHolder[] vertices = new VertexHolder[3];

80     public FaceHolder(VertexHolder v1, VertexHolder v2, VertexHolder v3) {
81         vertices[0] = v1;
82         vertices[1] = v2;
83         vertices[2] = v3;
84     }

86     protected LinkedList<FaceHolder> splitFace( final VertexHolder v, final
ArrayList<EdgeHolder> e ) {
87         for ( VertexHolder vertex: vertices )
88             e.add( new EdgeHolder( vertex, v ) );
89         final LinkedList<FaceHolder> faces = new LinkedList<FaceHolder>();
90         faces.addTail( new FaceHolder( vertices[0], vertices[1], v ) );
91         faces.addTail( new FaceHolder( vertices[1], vertices[2], v ) );
92         faces.addTail( new FaceHolder( vertices[2], vertices[0], v ) );
93         return faces;

```

```

94     }
95 }
96 }

```

D.3.2 mgtaylor.graph.graphTypes.PlanarMultiWheelGraph

```

1 package mgtaylor.graph.graphTypes;
2
3 import java.util.ArrayList;
4
5 import mgtaylor.graph.Graph;
6 import mgtaylor.graph.Vertex;
7
8 /**
9  * <p>A planar graph shaped like a wheel. By varying the number of spokes in the
10  * graph the maximum degree of the vertices in the graph can be controlled.</p>
11  *
12  * <p>The graph is formed from a central vertex; emanating from this are a
13  * number of spokes, each spoke formed of a straight line of vertices and
14  * connecting edges formed into a regular radial pattern around the central
15  * vertex; and in the centre of each triangular region formed by pairs of
16  * neighbouring spokes is a single (hub) vertex connected to each vertex in
17  * both sets of spokes and the central vertex. If the graph is formed with an
18  * outer ring of edges then the extreme vertex on each spoke is connected to
19  * the corresponding vertex terminating the neighbouring spokes and forming a
20  * ring around the graph.</p>
21  *
22  * <p>For a graph with  $v$  vertices and  $s$  spokes, there is:</p>
23  * <ul>
24  * <li>One central vertex, connected to  $2s$  vertices (each of the  $s$  hub vertices
25  * and the first vertex of each of the  $s$  spokes);</li>
26  * <li> $(v - s - 1)$  vertices forming the spokes; of these there are:
27  * <ul>
28  * <li> $W$  vertices terminating each spoke, each is connected 3 vertices if there
29  * is no outer ring of edges (the previous spoke vertex, or the centre vertex
30  * if the spoke is a single vertex long, and the two adjacent hub vertices) or
31  * 5 vertices if there is an outer ring;</li>
32  * <li> $(v - 2s - 1)$  vertices split evenly between  $s$  spokes, each is connected
33  * to 4 vertices (the two adjacent hub vertices and the next and previous
34  * vertex in the spoke)</li>
35  * </ul></li>
36  * <li> $W$  hub vertices, each is connected to  $2(v - s - 1)/s + 1$  vertices (each
37  * vertex in two spokes and the central vertex.</li>
38  * </ul>
39  *
40  * <p>A maximal planar graph will have  $(3v - 6)$  edges. A PlanarMultiWheelGraph
41  * with an outer ring has
42  *
43  *  $(2s + 5 + 4(v - 2s - 1) + s(2(v - s - 1)/s + 1) = (3v - 6 + s)$  edges. Therefore a PlanarMultiWheel graph
44  * with 3 spokes will be maximal planar and as the number of spokes increases
45  * there is a linear decrease in the number of edges compared to a maximal
46  * planar graph.</p>
47  *
48  * @author Martyn Taylor
49  *
50  */
51 public class PlanarMultiWheelGraph extends Graph {
52     private static final boolean SHOW_VERTEX_TYPE_IN_NAME = false;
53     private final Vertex center;
54     private final ArrayList<Vertex> hubs;
55     private final ArrayList<ArrayList<Vertex>> spokes;

```

```

57  /**
58  * <p>Creates a planar graph with the given number of vertices; the graph
59  * is formed from a number of radial spokes and each pair of adjacent
60  * spokes is connected by edges to a hub vertex forming a number of
61  * wheels.</p>
62  *
63  * @param numberOfVertices
64  * @param numberOfSpokes
65  * @param createOuterRing
66  */
67  public PlanarMultiWheelGraph( final int numberOfVertices, final int
numberOfSpokes, final boolean createOuterRing ) {
68      /*
69      * Number of Spokes is limited by (number of vertices - 1)/2.
70      * Wheel graph is maximal planar with the exception of the outer face.
71      * If there is an outer ring of edges connecting the vertices on the
72      * extreme of the spokes then there are (numberOfSpokes - 3) edges
73      * removed from a maximal planar graph; otherwise there is the outer
74      * ring of edges missing as well then there are an additional
75      * (numberOfSpokes) missing.
76      */
77      super( numberOfVertices,
78            3 * numberOfVertices - 3
79            - (createOuterRing ? 1 : 2) * Math.max( 3, Math.min( numberOfSpokes,
getMaximumNumberOfSpokes( numberOfVertices ) ) )
80            );
81      if ( numberOfVertices < 7 ) {
82          center = null;
83          hubs = new ArrayList<Vertex>( 0 );
84          spokes = new ArrayList<ArrayList<Vertex>>( 0 );
85          return;
86      }
87      final int wheels = Math.max( 3, Math.min( numberOfSpokes,
getMaximumNumberOfSpokes( numberOfVertices ) ) );
88      final int numberOfSpokeVertices = numberOfVertices - 1 - wheels;
89      final int baseSpokeLength = numberOfSpokeVertices / wheels;
90      final int additionalSpokes = numberOfSpokeVertices % wheels;
91      center = createVertex( 0, (SHOW_VERTEX_TYPE_IN_NAME ? "C1": "1"), wheels *
2 );
92      hubs = new ArrayList<Vertex>( wheels );
93      spokes = new ArrayList<ArrayList<Vertex>>( wheels );

94
95      int count = 2;
96      for ( int i = 0; i < wheels; i++ ) {
97          final int spokeLength = baseSpokeLength + (additionalSpokes > i ? 1 : 0);
98          final int nextSpokeLength = baseSpokeLength + (additionalSpokes > ((i +
wheels - 1) % wheels) ? 1 : 0);
99          Vertex hub = createVertex( count - 1, (SHOW_VERTEX_TYPE_IN_NAME ? "H": "")
+ Integer.toString( count ), spokeLength + nextSpokeLength + 1 );
100          count++;
101          hubs.add( hub );
102          createEdge( center, hub );
103          final ArrayList<Vertex> spoke = new ArrayList<Vertex>( spokeLength );
104          spokes.add( spoke );
105          Vertex previous = center;
106          for ( int s = 0; s < spokeLength; s++ ) {
107              Vertex v = createVertex( count - 1,
(SHOW_VERTEX_TYPE_IN_NAME ? "S": "") + Integer.toString( count ), 3 + (
s < spokeLength - 1 ? 1 : ( createOuterRing ? 2 : 0 ) ) );
108              count++;
109              spoke.add( v );
110              createEdge( previous, v );
111              createEdge( hub, v );
112              previous = v;

```



```

113     }
114 }
115 ArrayList<Vertex> prevSpoke = spokes.get( spokes.size() - 1 );
116 for ( int i = 0; i < wheels; i++ ) {
117     final ArrayList<Vertex> spoke = spokes.get( i );
118     final Vertex hub = hubs.get( ( i + 1 ) % wheels );
119     for ( Vertex v: spoke )
120         createEdge( hub, v );
121     if ( createOuterRing )
122         createEdge( spoke.get( spoke.size() - 1 ), prevSpoke.get(
123             prevSpoke.size() - 1 ) );
124     prevSpoke = spoke;
125 }
126
127 /**
128  * Returns the central vertex of the graph.
129  * @return Vertex
130  */
131 public Vertex getCenterVertex() {
132     return center;
133 }
134
135 /**
136  * Returns the hub vertices.
137  * @return
138  */
139 public ArrayList<Vertex> getHubs() {
140     return hubs;
141 }
142
143 /**
144  * Returns the spokes of the graph.
145  * @return
146  */
147 public ArrayList<ArrayList<Vertex>> getSpokes() {
148     return spokes;
149 }
150
151 /**
152  * Calculates the maximum number of spokes for the given number of
153  * vertices.
154  * @param numberOfVertices
155  * @return int
156  */
157 public static int getMaximumNumberOfSpokes( final int numberOfVertices ) {
158     return (numberOfVertices - 1) / 2;
159 }
160 }

```

D.3.3 mgtaylor.graph.graphTypes.PlanarWheelGraph

```

1 package mgtaylor.graph.graphTypes;
2
3 import java.util.ArrayList;
4
5 import mgtaylor.graph.Graph;
6 import mgtaylor.graph.Vertex;
7
8 public class PlanarWheelGraph extends Graph {
9     private final Vertex innerHub;
10    private final Vertex outerHub;

```

```

11 private final ArrayList<Vertex> rimVertices;

13 public PlanarWheelGraph( final int numberOfVertices ) {
14     super( numberOfVertices, (numberOfVertices < 2?0:(numberOfVertices ==
15         2?1:3 * numberOfVertices - 6)) );
16     final int numberOfRimVertices = numberOfVertices > 4?numberOfVertices - 2:
17     Math.min( numberOfVertices, 3 );
18     rimVertices = new ArrayList<Vertex>( numberOfRimVertices );

19     for ( int i = 1; i <= numberOfRimVertices; i++ )
20         rimVertices.add( createVertex( i - 1, Integer.toString( i ), 4 ) );
21     for ( int i = 1; i < numberOfRimVertices; i++ )
22         createEdge( rimVertices.get( i - 1 ), rimVertices.get( i ) );
23     outerHub = numberOfVertices > 4?createVertex( numberOfVertices - 2,
24     Integer.toString( numberOfVertices - 1 ), numberOfVertices - 1 ):null;
25     innerHub = numberOfVertices > 3?createVertex( numberOfVertices - 1,
26     Integer.toString( numberOfVertices ), numberOfVertices - 1 ):null;
27     if ( innerHub != null )
28         for ( int i = 0; i < numberOfRimVertices; i++ )
29             createEdge( innerHub, rimVertices.get( i ) );
30     if ( outerHub != null )
31         for ( int i = 0; i < numberOfRimVertices; i++ )
32             createEdge( outerHub, rimVertices.get( i ) );
33     if ( numberOfRimVertices > 2 )
34         createEdge( rimVertices.get( 0 ), rimVertices.get( numberOfRimVertices
35         - 1 ) );
36 }

37 public Vertex getInnerHubVertex() { return innerHub; }
38 public Vertex getOuterHubVertex() { return outerHub; }
39 public ArrayList<Vertex> getRimVertices() { return rimVertices; }
40 }

```

D.3.4 mgtaylor.graph.graphTypes.PermutableFlippableGraph

```

1 package mgtaylor.graph.graphTypes;

3 import mgtaylor.graph.Vertex;
4 import mgtaylor.graph.Graph;

6 public class PermutableFlippableGraph extends Graph {
7     public PermutableFlippableGraph( final int facets ) {
8         super( 3 + 2*(facets < 1?1:facets), 2 + 5*(facets < 1?1:facets) );
9         final int f = (facets < 1?1:facets);
10        final Vertex v0 = createVertex( 0, "1", 2 );
11        final Vertex v1 = createVertex( 1, "2", 2*f + 1 );
12        final Vertex v2 = createVertex( 2, "3", 2*f + 1 );
13        createEdge( v0, v1 );
14        createEdge( v2, v0 );
15        Vertex v = null;
16        for ( int i = 3; i < 3 + 2*f; i++ ) {
17            final Vertex vn = createVertex( i, Integer.toString( i + 1 ), 3 );
18            createEdge( v1, vn );
19            createEdge( v2, vn );
20            if ( i % 2 == 0 )
21                createEdge( v, vn );
22            v = vn;
23        }
24    }

26    public static void main( String[] args ) {
27        Graph g = new PermutableFlippableGraph( 200 );

```

```

28     System.out.println( g.toString() );
29     System.out.println( g.getVertices().size() + ", " + g.getEdges().size() );
30 }
31 }

```

D.3.5 mgtaylor.graph.graphTypes.RandomMaximalPlanarGraphGenerator

```

1 package mgtaylor.graph.graphTypes;
2
3 import java.util.ArrayList;
4
5 import mgtaylor.datastructures.LinkList;
6 import mgtaylor.graph.Edge;
7 import mgtaylor.graph.Graph;
8 import mgtaylor.graph.Vertex;
9
10 public class RandomMaximalPlanarGraphGenerator {
11     private VertexHolder[] vhs;
12     private EdgeHolder[] ehs;
13
14     public RandomMaximalPlanarGraphGenerator( final int size ) {
15         if ( size < 3 )
16             throw new IllegalArgumentException( "Size must be at least 3." );
17         vhs = new VertexHolder[ size ];
18         ehs = new EdgeHolder[ 3 * size - 6 ];
19         final ArrayList<VertexHolder[]> fhs = new ArrayList<VertexHolder[]>( 2 *
20             size - 5 );
21
22         for ( int i = 0; i < size; i++ )
23             vhs[i] = new VertexHolder( i );
24         for ( int i = 0; i < 3; i++ )
25             ehs[i] = new EdgeHolder( vhs[i], vhs[(i+1)%3] );
26         fhs.add( new VertexHolder[] { vhs[0], vhs[1], vhs[2] } );
27
28         for ( int i = 3; i < size; i++ ) {
29             int faceIndex = Math.min((int) Math.floor( Math.random() * (2 * i - 5)),
30                 2 * i - 6 );
31             VertexHolder[] face = fhs.get(faceIndex);
32             fhs.set( faceIndex, new VertexHolder[] { face[0], face[1], vhs[i] } );
33             fhs.add( new VertexHolder[] { face[1], face[2], vhs[i] } );
34             fhs.add( new VertexHolder[] { face[2], face[0], vhs[i] } );
35             for ( int j = 0; j < 3; j++ )
36                 ehs[3 * i - 6 + j] = new EdgeHolder( face[j], vhs[i] );
37         }
38     }
39
40     public Graph generateGraph( final int size ) {
41         if ( size < 3 )
42             throw new IllegalArgumentException( "Size must be at least 3." );
43         return new RandomMaximalPlanarGraph( size );
44     }
45
46     private class RandomMaximalPlanarGraph extends Graph {
47         private RandomMaximalPlanarGraph( int size ) {
48             super( size, 3 * size - 6 );
49             for ( int i = 0; i < size; i++ )
50                 this.getVertices().add( vhs[i].generateVertex( this, size ) );
51             for ( int i = 0; i < 3 * size - 6; i++ )
52                 this.getEdges().add( ehs[i].generateEdge( this, i ) );
53         }
54     }
55 }

```

```

54 private class VertexHolder {
55     private final int index;
56     private final LinkedList<EdgeHolder> edges = new LinkedList<EdgeHolder>();

58     private VertexHolder( final int index ) {
59         this.index = index;
60     }

62     private void connectEdge( final EdgeHolder edge ) {
63         edges.addTail( edge );
64     }

66     private Vertex generateVertex( final Graph graph, int size ) {
67         int e = 0;
68         for (EdgeHolder eh: edges)
69             if ( eh.from.index < size && eh.to.index < size )
70                 e++;
71         return new Vertex( graph, index, Integer.toString(index + 1), e );
72     }

74     private int getIndex() { return index; }
75 }

77 private class EdgeHolder {
78     private final VertexHolder from;
79     private final VertexHolder to;

81     private EdgeHolder( final VertexHolder f, final VertexHolder t ) {
82         from = f;
83         to = t;
84         from.connectEdge( this );
85         to.connectEdge( this );
86     }

88     public Edge generateEdge( final Graph graph, final int index ) {
89         final java.util.ArrayList<Vertex> vertices = graph.getVertices();
90         return new Edge( graph, vertices.get( from.getIndex() ), vertices.get(
91             to.getIndex() ), index );
92     }

94     public static void main( String[] args ) {
95         RandomMaximalPlanarGraphGenerator gg = new
96             RandomMaximalPlanarGraphGenerator( 10 );
97         for ( int i = 3; i <= 10; i++ )
98             System.out.println( gg.generateGraph( i ).toString() );
99     }

```

D.3.6 mgtaylor.graph.graphTypes.ReduceableGraphGenerator

```

1 package mgtaylor.graph.graphTypes;

3 import mgtaylor.datastructures.LinkedList;
4 import mgtaylor.graph.Graph;
5 import mgtaylor.graph.Vertex;
6 import mgtaylor.graph.Edge;

8 public class ReduceableGraphGenerator {
9     private final Vertex root;
10    private final int[] removableEdges;
11    private final VertexHolder[] v;

```

```

12     private final EdgeHolder[] e;

14     public ReduceableGraphGenerator( final Graph graph, final Vertex root ) {
15         this.root = root;
16         graph.DFS(root);
17         v = new VertexHolder[ graph.vertexSize() ];
18         e = new EdgeHolder[ graph.edgeSize() ];

20         for ( final Vertex vertex: graph.getVertices() )
21             v[vertex.getIndex()] = new VertexHolder( vertex.getIndex(),
22                 vertex.toString() );

23         int r = 0;
24         for ( final Edge edge: graph.getEdges() ) {
25             e[ edge.getIndex() ] = new EdgeHolder( v[ edge.getFrom().getIndex() ],
26                 v[ edge.getTo().getIndex() ] );
27             if ( edge.isDFSFrond() )
28                 r++;
29         }

30         removableEdges = new int[r];
31         r = 0;
32         for ( final Edge edge: graph.getEdges() )
33             if ( edge.isDFSFrond() )
34                 removableEdges[r++] = edge.getIndex();
35     }

37     public int numberOfRemovableEdges() {
38         return removableEdges.length;
39     }

41     public Graph generateGraph( final int NumberOfRemovedEdges ) {
42         int nre;
43         if ( NumberOfRemovedEdges < 0 )
44             nre = 0;
45         else if ( NumberOfRemovedEdges > removableEdges.length )
46             nre = removableEdges.length;
47         else
48             nre = NumberOfRemovedEdges;
49         return new ReduceableGraph( nre );
50     }

52     private class ReduceableGraph extends Graph {
53         private ReduceableGraph( final int NumberOfRemovedEdges ) {
54             super( v.length, NumberOfRemovedEdges );

56             for ( VertexHolder vh: v )
57                 this.getVertices().add( vh.generateVertex( this ) );

59             this.setDFSRoot( this.getVertices().get( root.getIndex() ) );

61             int i = 0;
62             int r = 0;
63             for ( EdgeHolder eh: e ) {
64                 if ( r < NumberOfRemovedEdges && removableEdges[r] == i )
65                     r++;
66                 else
67                     this.edges.add( eh.generateEdge( this, i - r ) );
68                 i++;
69             }
70         }
71     }

73     protected class VertexHolder {
74         private final String text;

```

```

75     private final int index;
76     private final LinkedList<EdgeHolder> edges = new LinkedList<EdgeHolder>();

78     protected VertexHolder( final int index, final String text ) {
79         this.index = index;
80         this.text = text;
81     }

83     protected void connectEdge( final EdgeHolder edge ) {
84         edges.addTail( edge );
85     }

87     protected void disconnectEdge( final EdgeHolder edge ) {
88         edges.remove( edge );
89     }

91     protected Vertex generateVertex( final Graph graph ) {
92         return new Vertex( graph, index, text, edges.size() );
93     }

95     protected int getIndex() { return index; }
96 }

98 protected class EdgeHolder {
99     private final VertexHolder from;
100    private final VertexHolder to;

102    protected EdgeHolder( final VertexHolder f, final VertexHolder t ) {
103        from = f;
104        to = t;
105        from.connectEdge( this );
106        to.connectEdge( this );
107    }

109    protected Edge generateEdge( final Graph graph, final int index ) {
110        final java.util.ArrayList<Vertex> vertices = graph.getVertices();
111        return new Edge( graph, vertices.get( from.getIndex() ), vertices.get(
112            to.getIndex() ), index );
113    }

115    public static void main( final String[] args ) throws Exception {
116        final int size = 50;
117        int count = 0;
118        while ( true ) {
119            final RandomMaximalPlanarGraphGenerator rg = new
120                RandomMaximalPlanarGraphGenerator( size );
121            final Graph gr = rg.generateGraph( size );
122            final ReduceableGraphGenerator gg = new ReduceableGraphGenerator( gr,
123                gr.getVertices().get(0) );
124            for ( int i = 0; i < gg.numberOfRemovableEdges(); i++ ) {
125                final Graph g = gg.generateGraph( i );
126                try {
127                    g.silentPlanarityTest();
128                    if ( !g.isPlanar() ) {
129                        System.out.println( g.toString() );
130                        return;
131                    }
132                } catch ( Exception e ) {
133                    System.out.println( g.toString() );
134                    throw e;
135                }
136            }
137            count++;
138            if ( count % 10000 == 0 )

```

```

137         System.out.println( count );
138     }
139     /**
140     /*
141         final String[] graphs = {
142             "nodes { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
143             17, 18, 19 }\nedges\n{\n(0, 1), (1, 2), (2, 3), (3, 4), (3, 5), (5,
144             6), (6, 7), (3, 8), (7, 9), (8, 10), (8, 11), (5, 12), (1, 12), (6,
145             12), (8, 13), (2, 13), (10, 13), (7, 14), (2, 14), (9, 14), (5, 15),
146             (1, 15), (12, 15), (3, 16), (0, 16), (4, 16), (2, 17), (5, 17), (6,
147             17), (3, 18), (0, 18), (16, 18), (12, 19), (5, 19), (15, 19)\n}",
148             "nodes { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
149             17, 18, 19 }\nedges\n{\n(0, 1), (1, 2), (2, 3), (3, 4), (3, 5), (4,
150             6), (4, 7), (5, 8), (1, 9), (2, 9), (7, 9), (4, 10), (2, 10), (6,
151             10), (2, 11), (6, 11), (10, 11), (10, 12), (2, 12), (11, 12), (3,
152             13), (4, 13), (6, 13), (6, 14), (4, 14), (10, 14), (6, 15), (10,
153             15), (11, 15), (4, 16), (10, 16), (14, 16), (1, 17), (3, 17), (5,
154             17), (3, 18), (4, 18), (13, 18), (6, 19), (10, 19), (15, 19)\n}",
155             "nodes { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
156             17, 18, 19 }\nedges\n{\n(0, 1), (1, 2), (2, 3), (3, 4), (3, 5), (2,
157             6), (4, 6), (2, 7), (4, 7), (6, 7), (4, 8), (1, 8), (6, 8), (3, 9),
158             (0, 9), (5, 9), (5, 10), (3, 10), (9, 10), (3, 11), (9, 11), (10,
159             11), (6, 12), (4, 12), (8, 12), (9, 13), (10, 13), (11, 13), (2,
160             14), (3, 14), (4, 14), (8, 15), (6, 15), (12, 15), (4, 16), (6, 16),
161             (7, 16), (1, 17), (6, 17), (8, 17), (3, 18), (4, 18), (14, 18), (3,
162             19), (0, 19), (9, 19)\n}",
163             "nodes { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
164             17, 18, 19 }\nedges\n{\n(0, 1), (1, 2), (2, 3), (3, 4), (3, 5), (4,
165             6), (3, 7), (4, 8), (8, 9), (6, 10), (6, 11), (10, 12), (9, 13),
166             (12, 14), (7, 15), (13, 16), (7, 17), (18, 3), (10, 18), (19, 3),
167             (19, 0), (11, 19)\n}",
168             "nodes { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
169             17, 18, 19 }\nedges\n{\n(0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (5,
170             6), (4, 7), (6, 8), (9, 2), (7, 9), (10, 6), (10, 5), (8, 10), (11,
171             0), (11, 1), (3, 11), (12, 2), (12, 4), (5, 12), (13, 2), (13, 3),
172             (7, 13), (14, 6), (14, 5), (10, 14), (15, 1), (15, 3), (11, 15),
173             (16, 2), (16, 7), (9, 16), (17, 2), (17, 7), (16, 17), (18, 4), (18,
174             2), (9, 18), (19, 10), (19, 6), (14, 19)\n}",
175             "nodes { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
176             17, 18, 19 }\nedges\n{\n(0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (3,
177             6), (4, 7), (7, 8), (8, 9), (8, 10), (6, 11), (7, 12), (6, 13), (5,
178             14), (6, 15), (7, 16), (13, 17), (18, 1), (15, 18), (19, 7), (19,
179             8), (10, 19)\n}",
180             "nodes { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
181             17, 18, 19 }\nedges\n{\n(0, 1), (1, 2), (2, 3), (3, 4), (3, 5), (5,
182             6), (5, 7), (4, 8), (8, 9), (7, 10), (8, 11), (11, 12), (9, 13),
183             (14, 11), (14, 3), (12, 14), (15, 2), (15, 0), (3, 15), (16, 3),
184             (16, 2), (15, 16), (17, 12), (17, 11), (14, 17), (18, 5), (18, 3),
185             (10, 18), (19, 3), (19, 12), (14, 19)\n}",
186             "nodes { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
187             17, 18, 19 }\nedges\n{\n(0, 1), (1, 2), (2, 3), (3, 4), (3, 5), (4,
188             6), (5, 7), (3, 8), (6, 9), (5, 10), (11, 2), (4, 11), (12, 2), (12,
189             0), (5, 12), (13, 2), (13, 0), (12, 13), (14, 0), (14, 5), (12, 14),
190             (15, 3), (15, 5), (7, 15), (16, 2), (16, 0), (13, 16), (17, 5), (17,
191             0), (7, 17), (18, 4), (18, 1), (11, 18), (19, 1), (19, 3), (8,
192             19)\n}"
193         };
194     }
195     final Graph g = new Graph( graphs[graphs.length - 1] );
196     g.setDFSRoot( g.getVertices().get(0) );
197     g.DFS();
198     for ( final Vertex v : g.getVertices() ) {
199         System.out.println( v + "\t(" + v.getDFSIndex() + ")\t" +
200             v.getLowPoint1() + ",\t" + v.getLowPoint2() );
201     }
202     g.generateSegments();
203     for ( mgttaylor.graph.Segment s : g.getSegments() ) {

```

```

158         s.setDepthFromParent();
159         for (int i = 0; i < s.getDepth(); i++)
160             System.out.print( " " );
161         System.out.print( s.toString() );
162         System.out.print( ", " );
163         System.out.println( s.getLeafVertex() );
164     }
165     g.embedSegments();
166     g.generateEdgeSides();
167     System.out.println( g.toVertexCyclicOrderString() );
168     /**/
169     }
170 }

```

D.3.7 mgtaylor.graph.graphTypes.ReduceableBiconnectedGraphGenerator

```

1 package mgtaylor.graph.graphTypes;

3 import mgtaylor.datastructures.LinkList;
4 import mgtaylor.graph.Graph;
5 import mgtaylor.graph.Vertex;
6 import mgtaylor.graph.Edge;
7 import mgtaylor.graph.Segment;

9 public class ReduceableBiconnectedGraphGenerator {
10     private final Vertex root;
11     private final int[] removableEdges;
12     private final VertexHolder[] v;
13     private final EdgeHolder[] e;

15     public ReduceableBiconnectedGraphGenerator( final Graph graph, final Vertex
        root ) {
16         this.root = root;
17         graph.DFS(root);
18         graph.generateSegments();
19         v = new VertexHolder[ graph.vertexSize() ];
20         e = new EdgeHolder[ graph.edgeSize() ];

22         for ( final Vertex vertex: graph.getVertices() )
23             v[ vertex.getIndex() ] = new VertexHolder( vertex.getIndex(),
                vertex.toString() );

25         for ( final Edge edge: graph.getEdges() )
26             e[ edge.getIndex() ] = new EdgeHolder( v[ edge.getFrom().getIndex() ],
                v[ edge.getTo().getIndex() ] );

28         int r = 0;
29         for ( final Segment segment: graph.getSegments() ) {
30             if ( segment.getEdges().size() == 1 ) {
31                 // System.out.println( segment + ", " +
32                 segment.getEdges().getTail().getIndex() );
33                 r++;
34             }
35         }

36         removableEdges = new int[ r ];
37         r = 0;
38         for ( final Segment segment: graph.getSegments() ) {
39             if ( segment.getEdges().size() == 1 ) {
40                 removableEdges[ r ] = segment.getEdges().getTail().getIndex();
41                 // System.out.print( removableEdges[ r ] + ", " );
42                 r++;

```



```

43     }
44 }
45 java.util.Arrays.sort( removableEdges );
46
47 // System.out.println( (removableEdges.length == r) );
48 }
49
50 public int numberOfRemovableEdges() {
51     return removableEdges.length;
52 }
53
54 public Graph generateGraph( final int NumberOfRemovedEdges ) {
55     int nre;
56     if ( NumberOfRemovedEdges < 0 )
57         nre = 0;
58     else if ( NumberOfRemovedEdges > numberOfRemovableEdges() )
59         nre = numberOfRemovableEdges();
60     else
61         nre = NumberOfRemovedEdges;
62     return new ReduceableGraph( nre );
63 }
64
65 private class ReduceableGraph extends Graph {
66     private ReduceableGraph( final int NumberOfRemovedEdges ) {
67         super( v.length, NumberOfRemovedEdges );
68
69         for ( VertexHolder vh: v )
70             this.getVertices().add( vh.generateVertex( this ) );
71
72         this.setDFSRoot( this.getVertices().get( root.getIndex() ) );
73
74         int i = 0;
75         int r = 0;
76         for ( EdgeHolder eh: e ) {
77             if ( r < NumberOfRemovedEdges && removableEdges[r] == i )
78                 r++;
79             else
80                 this.edges.add( eh.generateEdge( this, i - r ) );
81             i++;
82         }
83     }
84 }
85
86 protected class VertexHolder {
87     private final String text;
88     private final int index;
89     private final LinkedList<EdgeHolder> edges = new LinkedList<EdgeHolder>();
90
91     protected VertexHolder( final int index, final String text ) {
92         this.index = index;
93         this.text = text;
94     }
95
96     protected void connectEdge( final EdgeHolder edge ) {
97         edges.addTail( edge );
98     }
99
100    protected void disconnectEdge( final EdgeHolder edge ) {
101        edges.remove( edge );
102    }
103
104    protected Vertex generateVertex( final Graph graph ) {
105        return new Vertex( graph, index, text, edges.size() );
106    }

```

```

108     protected int getIndex() { return index; }
109 }

111 protected class EdgeHolder {
112     private final VertexHolder from;
113     private final VertexHolder to;

115     protected EdgeHolder( final VertexHolder f, final VertexHolder t ) {
116         from = f;
117         to = t;
118         from.connectEdge( this );
119         to.connectEdge( this );
120     }

122     protected Edge generateEdge( final Graph graph, final int index ) {
123         final java.util.ArrayList<Vertex> vertices = graph.getVertices();
124         return new Edge( graph, vertices.get( from.getIndex() ), vertices.get(
125             to.getIndex() ), index );
126     }

128     public static void main( final String[] args ) throws Exception {
129         final int size = 15000;
130         final Graph gr;
131         switch ( 3 ) {
132             case 2: gr = new TriangularPrismMaximalPlanarGraph( size ); break;
133             case 3: gr = new OrderedFaceTrisectionGraph( size ); break;
134             case 4: gr = new RandomMaximalPlanarGraphGenerator( size
135                 ).generateGraph( size ); break;
136             default: gr = new PlanarMultiWheelGraph( size, 3, true ); break;
137         }

138         final ReduceableBiconnectedGraphGenerator gg = new
139             ReduceableBiconnectedGraphGenerator( gr, gr.getVertices().get(0) );
140         // final Graph g = gg.generateGraph( gg.numberOfRemovableEdges() );
141         // System.out.println( g.toString() );
142         for ( int i = 0; i < gg.numberOfRemovableEdges(); i += 100 ) {
143             final Graph g = gg.generateGraph( i );
144             g.DFS();
145             g.generateSegments();
146             g.embedSegments();
147             System.out.println( g.edgeSize() + "\t" +
148                 g.calculateAverageSegmentDepth() + "\t" + g.getNumberOfSegments() +
149                 "\t" + g.getPermutations().size() + "\t" +
150                 g.calculateNumberOfFlippableBlocks() );
151         }

152         /**/
153         /*
154         final int size = 20;
155         int count = 0;
156         while ( true ) {
157             final RandomMaximalPlanarGraphGenerator rg = new
158                 RandomMaximalPlanarGraphGenerator( size );
159             final Graph gr = rg.generateGraph( size );
160             final ReduceableGraphGenerator gg = new ReduceableGraphGenerator( gr,
161                 gr.getVertices().get(0) );
162             for ( int i = 0; i < gg.numberOfRemovableEdges(); i++ ) {
163                 final Graph g = gg.generateGraph( i );
164                 try {
165                     g.silentPlanarityTest();
166                     if ( !g.isPlanar() ) {
167                         System.out.println( g.toString() );
168                         return;
169                     }
170                 } catch ( Exception e ) {
171                     System.out.println( g.toString() );
172                 }
173             }
174             count++;
175         }
176     }

```

```

165         throw e;
166     }
167 }
168 count++;
169 if ( count % 100000 == 0 )
170     System.out.println( count );
171 }
172 /**/
173 /*     final String[] graphs = {
174     "nodes { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
175     17, 18, 19 }\nedges\n{\n(0, 1), (1, 2), (2, 3), (3, 4), (3, 5), (5,
176     6), (6, 7), (3, 8), (7, 9), (8, 10), (8, 11), (5, 12), (1, 12), (6,
177     12), (8, 13), (2, 13), (10, 13), (7, 14), (2, 14), (9, 14), (5, 15),
178     (1, 15), (12, 15), (3, 16), (0, 16), (4, 16), (2, 17), (5, 17), (6,
179     17), (3, 18), (0, 18), (16, 18), (12, 19), (5, 19), (15, 19)\n}",
180     "nodes { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
181     17, 18, 19 }\nedges\n{\n(0, 1), (1, 2), (2, 3), (3, 4), (3, 5), (4,
182     6), (4, 7), (5, 8), (1, 9), (2, 9), (2, 9), (7, 9), (4, 10), (2, 10), (6,
183     10), (2, 11), (6, 11), (10, 11), (10, 12), (2, 12), (11, 12), (3,
184     13), (4, 13), (6, 13), (6, 14), (4, 14), (10, 14), (6, 15), (10,
185     15), (11, 15), (4, 16), (10, 16), (14, 16), (1, 17), (3, 17), (5,
186     17), (3, 18), (4, 18), (13, 18), (6, 19), (10, 19), (15, 19)\n}",
187     "nodes { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
188     17, 18, 19 }\nedges\n{\n(0, 1), (1, 2), (2, 3), (3, 4), (3, 5), (2,
189     6), (4, 6), (2, 7), (4, 7), (6, 7), (4, 8), (1, 8), (6, 8), (3, 9),
190     (0, 9), (5, 9), (5, 10), (3, 10), (9, 10), (3, 11), (9, 11), (10,
191     11), (6, 12), (4, 12), (8, 12), (9, 13), (10, 13), (11, 13), (2,
192     14), (3, 14), (4, 14), (8, 15), (6, 15), (12, 15), (4, 16), (6, 16),
193     (7, 16), (1, 17), (6, 17), (8, 17), (3, 18), (4, 18), (14, 18), (3,
194     19), (0, 19), (9, 19)\n}",
195     "nodes { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
196     17, 18, 19 }\nedges\n{\n(0, 1), (1, 2), (2, 3), (3, 4), (3, 5), (4,
197     6), (3, 7), (4, 8), (8, 9), (6, 10), (6, 11), (10, 12), (9, 13),
198     (12, 14), (7, 15), (13, 16), (7, 17), (18, 3), (10, 18), (19, 3),
199     (19, 0), (11, 19)\n}",
200     "nodes { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
201     17, 18, 19 }\nedges\n{\n(0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (5,
202     6), (4, 7), (6, 8), (9, 2), (7, 9), (10, 6), (10, 5), (8, 10), (11,
203     0), (11, 1), (3, 11), (12, 2), (12, 4), (5, 12), (13, 2), (13, 3),
204     (7, 13), (14, 6), (14, 5), (10, 14), (15, 1), (15, 3), (11, 15),
205     (16, 2), (16, 7), (9, 16), (17, 2), (17, 7), (16, 17), (18, 4), (18,
206     2), (9, 18), (19, 10), (19, 6), (14, 19)\n}",
207     "nodes { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
208     17, 18, 19 }\nedges\n{\n(0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (3,
209     6), (4, 7), (7, 8), (8, 9), (8, 10), (6, 11), (7, 12), (6, 13), (5,
210     14), (6, 15), (7, 16), (13, 17), (18, 1), (15, 18), (19, 7), (19,
211     8), (10, 19)\n}",
212     "nodes { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
213     17, 18, 19 }\nedges\n{\n(0, 1), (1, 2), (2, 3), (3, 4), (3, 5), (5,
214     6), (5, 7), (4, 8), (8, 9), (7, 10), (8, 11), (11, 12), (9, 13),
215     (14, 11), (14, 3), (12, 14), (15, 2), (15, 0), (3, 15), (16, 3),
216     (16, 2), (15, 16), (17, 12), (17, 11), (14, 17), (18, 5), (18, 3),
217     (10, 18), (19, 3), (19, 12), (14, 19)\n}",
218     "nodes { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
219     17, 18, 19 }\nedges\n{\n(0, 1), (1, 2), (2, 3), (3, 4), (3, 5), (4,
220     6), (5, 7), (3, 8), (6, 9), (5, 10), (11, 2), (4, 11), (12, 2), (12,
221     0), (5, 12), (13, 2), (13, 0), (12, 13), (14, 0), (14, 5), (12, 14),
222     (15, 3), (15, 5), (7, 15), (16, 2), (16, 0), (13, 16), (17, 5), (17,
223     0), (7, 17), (18, 4), (18, 1), (11, 18), (19, 1), (19, 3), (8,
224     19)\n}"
225 };
226 final Graph g = new Graph( graphs[graphs.length - 1] );
227 g.setDFSRoot( g.getVertices().get(0) );
228 g.DFS();
229 for ( final Vertex v : g.getVertices() ) {

```

```

187         System.out.println( v + "\t(" + v.getDFSIndex() + ")\t" +
188                               v.getLowPoint1() + ",\t" + v.getLowPoint2() );
189     }
190     g.generateSegments();
191     for ( mgtaylor.graph.Segment s : g.getSegments() ) {
192         s.setDepthFromParent();
193         for ( int i = 0; i < s.getDepth(); i++ )
194             System.out.print( "  " );
195         System.out.print( s.toString() );
196         System.out.print( ", " );
197         System.out.println( s.getLeafVertex() );
198     }
199     g.embedSegments();
200     g.generateEdgeSides();
201     System.out.println( g.toVertexCyclicOrderString() );
202     /**/
203 }

```

D.3.8 mgtaylor.graph.graphTypes.TriangularPrismMaximalPlanarGraph

```

1 package mgtaylor.graph.graphTypes;
2
3 import mgtaylor.graph.Graph;
4 import mgtaylor.graph.Vertex;
5
6 public class TriangularPrismMaximalPlanarGraph extends Graph {
7
8     public TriangularPrismMaximalPlanarGraph( final int numberOfVertices ) {
9         super(Math.max(3, numberOfVertices), 3 * Math.max(3, numberOfVertices) - 6);
10    }
11    final int vertexSize = Math.max( 3, numberOfVertices );
12    final int extraVertices = vertexSize % 3;
13
14    final Vertex[] vs = new Vertex[ vertexSize ];
15    int i = 0;
16    if ( extraVertices > 0 ) {
17        vs[i] = createVertex( i, Integer.toString( i + 1 ), 3 );
18        i++;
19    }
20    for ( int k = 0; k < 3; k++ ) {
21        vs[i] = createVertex( i, Integer.toString( i + 1 ), vertexSize >
22                               5?(extraVertices > 0?5:4):(3 + extraVertices) );
23        for ( int j = 0; j < i; j++ )
24            createEdge( vs[j], vs[i] );
25        i++;
26    }
27    for ( ; i < vertexSize - (extraVertices == 2?1:0); i += 3 ) {
28        final int capacity = (i + 3) < vertexSize?6:(4 + (extraVertices ==
29                               2?1:0));
30        vs[i] = createVertex( i, Integer.toString( i + 1 ), capacity );
31        vs[i+1] = createVertex( i+1, Integer.toString( i + 2 ), capacity );
32        vs[i+2] = createVertex( i+2, Integer.toString( i + 3 ), capacity );
33        createEdge( vs[i], vs[i + 1] );
34        createEdge( vs[i], vs[i + 2] );
35        createEdge( vs[i + 1], vs[i + 2] );
36        createEdge( vs[i - 3], vs[i] );
37        createEdge( vs[i - 1], vs[i] );
38        createEdge( vs[i - 2], vs[i + 1] );
39        createEdge( vs[i - 3], vs[i + 1] );
40        createEdge( vs[i - 1], vs[i + 2] );
41        createEdge( vs[i - 2], vs[i + 2] );

```

```
39     }
40     if ( extraVertices == 2 ) {
41         vs[i] = createVertex( i, Integer.toString( i + 1 ), 3 );
42         for ( int j = i - 3; j < i; j++ )
43             createEdge( vs[j], vs[i] );
44     }
45 }
46 }
```

D.4 Empirical Testing Files

D.4.1 mgtaylor.graph.test.PlanaritySpeedTestViewer

```

1 package mgtaylor.test;

3 import java.io.*;
4 import java.util.*;

6 import javax.swing.*;

8 import mgtaylor.graph.*;
9 import mgtaylor.test.parsers.GraphExtractorLogParser;

11 public class PlanaritySpeedTestViewer extends JFrame {
12     private static final int graphRepetitions = 500;
13     private static final int graphMinSize = 20;
14     private static final int graphMaxSize = 5000;
15     private static final int graphSizeIncrement = 1;
16     private static final int graphMinWheels = 3;
17     private static final int graphMaxWheels = 3;
18     /**/
19     private static final long serialVersionUID = -556303577105329510L;

21     public PlanaritySpeedTestViewer() {
22         PrintStream writer = System.out;
23         for ( int wheels = graphMinWheels; wheels <= graphMaxWheels; wheels++ ) {
24             for ( int v = graphMaxSize; v >= graphMinSize; v -= graphSizeIncrement
25                 ) {
26                 Graph graph = new mgtaylor.graph.graphTypes.PlanarMultiWheelGraph(
27                     v, wheels, true );
28                 writer.println( graph.toString() );

29                 Random random = new Random();
30                 ArrayList<Vertex> vertices = graph.getVertices();
31                 for ( int r = 1; r <= graphRepetitions; r++ ) {
32                     int index = random.nextInt( v );
33                     Vertex root = vertices.get( index );
34                     graph.timePlanarityTest( root, writer );
35                 }
36                 graph = null;
37             }
38         }

40     public static void speedTestGraphWithDifferentRoot( final PrintStream out,
41         final Graph g, final int repetitions ) {
42         speedTestGraphWithDifferentRoot( out, g, repetitions, 0, g.vertexSize() -
43             1 );
44     }

46     public static int getRootIndex( int size, int numer, int denom ){
47         return Math.round( numer * ( size - 1 ) / denom );
48     }

49     public static void speedTestGraphWithDifferentRoot( final PrintStream out,
50         final Graph g, final int repetitions, final int start, final int end ) {
51         if ( out != null ) out.println( g.toString() );
52         for ( int i = start; i <= end; i++ ) {
53             final Vertex root = g.getVertices().get( i );
54             for ( int j = 0; j < repetitions; j++ )
55                 g.timePlanarityTest( root, out );
56         }
57     }

```

```

55     }

57     public static void speedTestGraphWithIncrementingWheels( final PrintStream
out, final int size, final int fromWheel, final int toWheel, final int
wheelIncrement, final int repetitions ) {
58         for ( int i = fromWheel; i <= toWheel; i += wheelIncrement ) {
59             final Graph g = new mgtaylor.graph.graphTypes.PlanarMultiWheelGraph(
size, i, true );
60             out.println( g.toString() );
61             for ( int r = 1; r <= repetitions; r++ )
62                 g.timePlanarityTest( out );
63         }
64     }

66     public static void speedTestSimilarTriangleGraphsWithIncreasingSizes( final
PrintStream out, final int from, final int to, final int increment, final int
repetitions, final int numerator, final int denominator ) {
67         for ( int i = from; i <= to; i += increment ) {
68             final Graph g = new
mgtaylor.graph.graphTypes.TriangularPrismMaximalPlanarGraph( i );
69             out.println( g.toString() );
70             final ArrayList<Vertex> vertices = g.getVertices();
71             final Vertex root = vertices.get( getRootIndex( vertices.size(),
numerator, denominator ) );
72             for ( int r = 0; r < repetitions; r++ )
73                 g.timePlanarityTest( root, out );
74         }
75     }

77     public static void speedTestSimilarWheelGraphsWithIncreasingSizes( final
PrintStream out, final int from, final int to, final int increment, final int
repetitions, final int numerator, final int denominator ) {
78         for ( int i = from; i <= to; i += increment ) {
79             final Graph g = new mgtaylor.graph.graphTypes.PlanarMultiWheelGraph( i,
3, true );
80             out.println( g.toString() );
81             final ArrayList<Vertex> vertices = g.getVertices();
82             final Vertex root = vertices.get( getRootIndex( vertices.size(),
numerator, denominator ) );
83             for ( int r = 0; r < repetitions; r++ )
84                 g.timePlanarityTest( root, out );
85         }
86     }

88     public static void speedTestSimilarOrderedGraphsWithIncreasingSizes( final
PrintStream out, final int from, final int to, final int increment, final int
repetitions, final int numerator, final int denominator ) {
89         for ( int i = from; i <= to; i += increment ) {
90             final Graph g = new
mgtaylor.graph.graphTypes.OrderedFaceTrisectionGraph( i );
91             out.println( g.toString() );
92             final ArrayList<Vertex> vertices = g.getVertices();
93             final Vertex root = vertices.get( getRootIndex( vertices.size(),
numerator, denominator ) );
94             for ( int r = 0; r < repetitions; r++ )
95                 g.timePlanarityTest( root, out );
96         }
97     }

99     public static void speedTestSimilarRandomGraphsWithIncreasingSizes(
100         final PrintStream out,
101         final mgtaylor.graph.graphTypes.RandomMaximalPlanarGraphGenerator gg,
102         final int from,
103         final int to,
104         final int increment,

```

```

105         final int repetitions ,
106         final int numerator ,
107         final int denominator
108     ) {
109         for ( int i = from; i <= to; i += increment ) {
110             final Graph g = gg.generateGraph( i );
111             out.println( g.toString() );
112             final ArrayList<Vertex> vertices = g.getVertices();
113             final Vertex root = vertices.get( getRootIndex( vertices.size(),
114                 numerator, denominator ) );
115             for ( int r = 0; r < repetitions; r++ )
116                 g.timePlanarityTest( root, out );
117         }
118     }
119
120     public static void speedTestGraphWithReducingEdges(
121         final PrintStream out,
122         final mgtaylor.graph.graphTypes.ReduceableBiconnectedGraphGenerator gg,
123         final int increment,
124         final int repetitions
125     ) {
126         performCodeWarmup( gg.generateGraph( gg.numberOfRemovableEdges() ),
127             CODE_WARMUP_REPETITIONS );
128         for ( int i = gg.numberOfRemovableEdges(); i >= 0; i -= increment ) {
129             final Graph g = gg.generateGraph( i );
130             out.println( g.toString() );
131             for ( int r = 0; r < repetitions; r++ )
132                 g.timePlanarityTest( null, out );
133         }
134     }
135
136     public static void performCodeWarmup( final Graph g, final int repetitions ) {
137         final ArrayList<Vertex> vertices = g.getVertices();
138         final Random random = new Random();
139         for ( int r = 0; r < repetitions; r++ )
140             g.silentPlanarityTest( vertices.get( random.nextInt( vertices.size() ) ) );
141         System.gc();
142     }
143
144     private static final int CODE_WARMUP_REPETITIONS = 50000;
145     private static final int REPETITIONS = 1000;
146     private static final int REPETITIONS_LARGE = 100;
147     private static final int REPETITIONS_ROOTS = 3000;
148
149     private static final int DIFFERENT_ROOTS = 0;
150     private static final int DIFFERENT_ROOTS_NUMBER_OF_TESTS = 10;
151     private static final int DIFFERENT_SIZES = DIFFERENT_ROOTS + 4 *
152         DIFFERENT_ROOTS_NUMBER_OF_TESTS;
153     private static final int DIFFERENT_SIZES_NUMBER_OF_TESTS = 5;
154     private static final int LARGE_SIZES = DIFFERENT_SIZES + 4 *
155         DIFFERENT_SIZES_NUMBER_OF_TESTS;
156     private static final int LARGE_SIZES_NUMBER_OF_TESTS = 5;
157     private static final int EDGE_SIZES = LARGE_SIZES + 4 *
158         LARGE_SIZES_NUMBER_OF_TESTS;
159     private static final int EDGE_SIZES_NUMBER_OF_TESTS = 5;
160
161     public static void performTestCase( final int testCase ) {
162         /**
163          * Test Case    Graph Tested
164          * -1          Reduceable Random Maximal Planar Graph (2500 vertices, 2500 -
165          * 4750 edges in 50 edge increments) - 1st Vertex As Root
166          * 0 - 9       Planar Wheel Graph (250-2500 Vertices in 250 increments) - All
167          *              Roots

```



```

162      * 10 - 19      Triangular Prism Graph (250-2500 Vertices in 250
increments) - All Roots
163      * 20 - 29      Ordered Maximal Planar Graph (250-2500 Vertices in 250
increments) - All Roots
164      * 30 - 39      Random Maximal Planar Graph (250-2500 Vertices in 250
increments) - All Roots
165      * 40          Planar Wheel Graph (40:3:4498 Vertices) - 1st Vertex as Root
166      * 41          Planar Wheel Graph (40:3:4498 Vertices) - Nth Vertex as Root
167      * 42          Planar Wheel Graph (40:12:4498 Vertices) - 1/4Nth Vertex as Root
168      * 43          Planar Wheel Graph (40:6:4498 Vertices) - 1/2Nth Vertex as Root
169      * 44          Planar Wheel Graph (40:12:4498 Vertices) - 3/4Nth Vertex as Root
170      * 45          Triangular Prism Graph (40:3:4498 Vertices) - 1st Vertex as
Root
171      * 46          Triangular Prism Graph (40:3:4498 Vertices) - Nth Vertex as
Root
172      * 47          Triangular Prism Graph (40:12:4498 Vertices) - 1/4Nth Vertex as
Root
173      * 48          Triangular Prism Graph (40:6:4498 Vertices) - 1/2Nth Vertex as
Root
174      * 49          Triangular Prism Graph (40:12:4498 Vertices) - 3/4Nth Vertex as
Root
175      * 50          Ordered Maximal Planar Graph (40:3:4498 Vertices) - 1st Vertex
as Root
176      * 51          Ordered Maximal Planar Graph (40:3:4498 Vertices) - Nth Vertex
as Root
177      * 52          Ordered Maximal Planar Graph (40:12:4498 Vertices) - 1/4Nth
Vertex as Root
178      * 53          Ordered Maximal Planar Graph (40:6:4498 Vertices) - 1/2Nth
Vertex as Root
179      * 54          Ordered Maximal Planar Graph (40:12:4498 Vertices) - 3/4Nth
Vertex as Root
180      * 55          Random Maximal Planar Graph (40:3:4498 Vertices) - 1st Vertex
as Root
181      * 56          Random Maximal Planar Graph (40:3:4498 Vertices) - Nth Vertex
as Root
182      * 57          Random Maximal Planar Graph (40:12:4498 Vertices) - 1/4Nth
Vertex as Root
183      * 58          Random Maximal Planar Graph (40:6:4498 Vertices) - 1/2Nth
Vertex as Root
184      * 59          Random Maximal Planar Graph (40:12:4498 Vertices) - 3/4Nth
Vertex as Root
185      * 60          Planar Wheel Graph (502:30:15000 Vertices) - 1st Vertex as Root
186      * 61          Planar Wheel Graph (502:30:15000 Vertices) - Nth Vertex as Root
187      * 62          Planar Wheel Graph (502:60:15000 Vertices) - 1/4Nth Vertex as
Root
188      * 63          Planar Wheel Graph (502:30:15000 Vertices) - 1/2Nth Vertex as
Root
189      * 64          Planar Wheel Graph (502:60:15000 Vertices) - 3/4Nth Vertex as
Root
190      * 65          Triangular Prism Graph (501:30:14999 Vertices) - 1st Vertex as
Root
191      * 66          Triangular Prism Graph (501:30:14999 Vertices) - Nth Vertex as
Root
192      * 67          Triangular Prism Graph (501:60:14999 Vertices) - 1/4Nth Vertex
as Root
193      * 68          Triangular Prism Graph (501:30:14999 Vertices) - 1/2Nth Vertex
as Root
194      * 69          Triangular Prism Graph (501:60:14999 Vertices) - 3/4Nth Vertex
as Root
195      * 70          Ordered Maximal Planar Graph (501:30:14999 Vertices) - 1st
Vertex as Root
196      * 71          Ordered Maximal Planar Graph (501:30:14999 Vertices) - Nth
Vertex as Root
197      * 72          Ordered Maximal Planar Graph (501:60:14999 Vertices) - 1/4Nth
Vertex as Root

```

```

198      * 73      Ordered Maximal Planar Graph (501:30:14999 Vertices) - 1/2Nth
      Vertex as Root
199      * 74      Ordered Maximal Planar Graph (501:60:14999 Vertices) - 3/4Nth
      Vertex as Root
200      * 75      Random Maximal Planar Graph (501:30:14999 Vertices) - 1st
      Vertex as Root
201      * 76      Random Maximal Planar Graph (501:30:14999 Vertices) - Nth
      Vertex as Root
202      * 77      Random Maximal Planar Graph (501:60:14999 Vertices) - 1/4Nth
      Vertex as Root
203      * 78      Random Maximal Planar Graph (501:30:14999 Vertices) - 1/2Nth
      Vertex as Root
204      * 79      Random Maximal Planar Graph (501:60:14999 Vertices) - 3/4Nth
      Vertex as Root
205      * 80      Planar Wheel Graph (15000 Vertices , Reducing Edges) - 1st
      Vertex as Root
206      * 81      Planar Wheel Graph (15000 Vertices , Reducing Edges) - Nth
      Vertex as Root
207      * 82      Planar Wheel Graph (15000 Vertices , Reducing Edges) - 1/4Nth
      Vertex as Root
208      * 83      Planar Wheel Graph (15000 Vertices , Reducing Edges) - 1/2Nth
      Vertex as Root
209      * 84      Planar Wheel Graph (15000 Vertices , Reducing Edges) - 3/4Nth
      Vertex as Root
210      * 85      Triangular Prism Graph (14999 Vertices , Reducing Edges) - 1st
      Vertex as Root
211      * 86      Triangular Prism Graph (14999 Vertices , Reducing Edges) - Nth
      Vertex as Root
212      * 87      Triangular Prism Graph (14999 Vertices , Reducing Edges) - 1/4
      Nth Vertex as Root
213      * 88      Triangular Prism Graph (14999 Vertices , Reducing Edges) - 1/2
      Nth Vertex as Root
214      * 89      Triangular Prism Graph (14999 Vertices , Reducing Edges) - 3/4
      Nth Vertex as Root
215      * 90      Ordered Maximal Planar Graph (14999 Vertices , Reducing Edges)
      - 1st Vertex as Root
216      * 91      Ordered Maximal Planar Graph (14999 Vertices , Reducing Edges)
      - Nth Vertex as Root
217      * 92      Ordered Maximal Planar Graph (14999 Vertices , Reducing Edges)
      - 1/4Nth Vertex as Root
218      * 93      Ordered Maximal Planar Graph (14999 Vertices , Reducing Edges)
      - 1/2Nth Vertex as Root
219      * 94      Ordered Maximal Planar Graph (14999 Vertices , Reducing Edges)
      - 3/4Nth Vertex as Root
220      * 95      Random Maximal Planar Graph (14999 Vertices , Reducing Edges) -
      1st Vertex as Root
221      * 96      Random Maximal Planar Graph (14999 Vertices , Reducing Edges) -
      Nth Vertex as Root
222      * 97      Random Maximal Planar Graph (14999 Vertices , Reducing Edges) -
      1/4Nth Vertex as Root
223      * 98      Random Maximal Planar Graph (14999 Vertices , Reducing Edges) -
      1/2Nth Vertex as Root
224      * 99      Random Maximal Planar Graph (14999 Vertices , Reducing Edges) -
      3/4Nth Vertex as Root
225      */
226      if ( testCase < 0 ) {
227          final Graph g;
228          final int size = 100;
229          switch ( testCase ) {
230              case -2:
231                  g = new mgtaylor.graph.graphTypes.TriangularPrismMaximalPlanarGraph(
                      size );
232                  break;
233              case -3:
234                  g = new mgtaylor.graph.graphTypes.OrderedFaceTrisectionGraph( size );

```

```

235         break;
236     case -4:
237         final mgtaylor.graph.graphTypes.RandomMaximalPlanarGraphGenerator gg
238             = new
239                 mgtaylor.graph.graphTypes.RandomMaximalPlanarGraphGenerator( size
240                     );
241         g = gg.generateGraph( size );
242         break;
243     case -1:
244     default:
245         g = new mgtaylor.graph.graphTypes.PlanarMultiWheelGraph( size , 3,
246             true );
247         break;
248     }
249     final mgtaylor.graph.graphTypes.ReduceableGraphGenerator rgg = new
250         mgtaylor.graph.graphTypes.ReduceableGraphGenerator( g,
251             g.getVertices().get( 0 ) );
252     performCodeWarmup( g, CODE_WARMUP_REPETITIONS );
253     final int steps = 100;
254     for ( int e = 0; e <= steps; e++ ) {
255         final Graph graph = rgg.generateGraph(
256             e*rgg.numberOfRemovableEdges()/steps );
257         System.out.println( graph.toString() );
258         for ( int i = 0; i < 0.5*REPETITIONS; i++ )
259             graph.timePlanarityTest();
260     }
261 } else if ( DIFFERENT_ROOTS <= testCase && testCase < DIFFERENT_SIZES ) {
262     final Graph g;
263     final int size;
264     final double repetition_factor;
265     switch ( (testCase - DIFFERENT_ROOTS) % DIFFERENT_ROOTS_NUMBER_OF_TESTS
266 ) {
267     case 0: size = 250; repetition_factor = 1; break;
268     case 1: size = 500; repetition_factor = 1; break;
269     case 2: size = 750; repetition_factor = 1; break;
270     case 3: size = 1000; repetition_factor = 1; break;
271     case 4: size = 1250; repetition_factor = 1; break;
272     case 5: size = 1500; repetition_factor = 1; break;
273     case 6: size = 1750; repetition_factor = 0.75; break;
274     case 7: size = 2000; repetition_factor = 0.5; break;
275     case 8: size = 2250; repetition_factor = 0.4; break;
276     case 9: size = 2500; repetition_factor = 0.3; break;
277     default: size = 1000; repetition_factor = 1; break;
278     }
279     switch ( (testCase - DIFFERENT_ROOTS) / DIFFERENT_ROOTS_NUMBER_OF_TESTS
280 ) {
281     case 1:
282         g = new mgtaylor.graph.graphTypes.TriangularPrismMaximalPlanarGraph(
283             size );
284         break;
285     case 2:
286         g = new mgtaylor.graph.graphTypes.OrderedFaceTrisectionGraph( size );
287         break;
288     case 3:
289         final mgtaylor.graph.graphTypes.RandomMaximalPlanarGraphGenerator gg
290             = new
291                 mgtaylor.graph.graphTypes.RandomMaximalPlanarGraphGenerator( size
292                     );
293         g = gg.generateGraph( size );
294         break;
295     case 0:
296     default:
297         g = new mgtaylor.graph.graphTypes.PlanarMultiWheelGraph( size , 3,
298             true );

```

```

288         break;
289     }
290     performCodeWarmup( g, CODE_WARMUP_REPETITIONS );
291     speedTestGraphWithDifferentRoot( System.out, g, (int)
292         (REPETITIONS_ROOTS * repetition_factor) );
293 } else if ( DIFFERENT_SIZES <= testCase && testCase < (LARGE_SIZES) ) {
294     final Graph g;
295     final int numer, denom, incr;
296     switch ( (testCase - DIFFERENT_SIZES) % DIFFERENT_SIZES_NUMBER_OF_TESTS ) {
297     case 0:      numer = 0; denom = 1; incr = 3; break;
298     case 1:      numer = 1; denom = 1; incr = 3; break;
299     case 2:      numer = 1; denom = 4; incr = 12; break;
300     case 3:      numer = 1; denom = 2; incr = 6; break;
301     case 4:      numer = 3; denom = 4; incr = 12; break;
302     default:     numer = 0; denom = 1; incr = 3; break;
303     }
304     switch ( (testCase - DIFFERENT_SIZES) / DIFFERENT_SIZES_NUMBER_OF_TESTS ) {
305     case 1:
306         g = new mgtaylor.graph.graphTypes.TriangularPrismMaximalPlanarGraph(
307             1000 );
308         performCodeWarmup( g, CODE_WARMUP_REPETITIONS );
309         speedTestSimilarTriangleGraphsWithIncreasingSizes( System.out, 42,
310             4500, incr, REPETITIONS, numer, denom );
311         break;
312     case 2:
313         g = new mgtaylor.graph.graphTypes.OrderedFaceTrisectionGraph( 1000 );
314         performCodeWarmup( g, CODE_WARMUP_REPETITIONS );
315         speedTestSimilarOrderedGraphsWithIncreasingSizes( System.out, 42,
316             4500, incr, REPETITIONS, numer, denom );
317         break;
318     case 3:
319         final mgtaylor.graph.graphTypes.RandomMaximalPlanarGraphGenerator gg
320             = new
321             mgtaylor.graph.graphTypes.RandomMaximalPlanarGraphGenerator( 4500
322             );
323         g = gg.generateGraph(1000);
324         performCodeWarmup( g, CODE_WARMUP_REPETITIONS );
325         speedTestSimilarRandomGraphsWithIncreasingSizes( System.out, gg, 42,
326             4500, incr, REPETITIONS, numer, denom );
327         break;
328     case 0:
329     default:
330         g = new mgtaylor.graph.graphTypes.PlanarMultiWheelGraph( 1000, 3,
331             true );
332         performCodeWarmup( g, CODE_WARMUP_REPETITIONS );
333         speedTestSimilarWheelGraphsWithIncreasingSizes( System.out, 40,
334             4498, incr, REPETITIONS, numer, denom );
335         break;
336     }
337 } else if ( LARGE_SIZES <= testCase && testCase < EDGE_SIZES ) {
338     final Graph g;
339     final int numer, denom, incr;
340     switch ( (testCase - LARGE_SIZES) % LARGE_SIZES_NUMBER_OF_TESTS ) {
341     case 0:      numer = 0; denom = 1; incr = 30; break;
342     case 1:      numer = 1; denom = 1; incr = 30; break;
343     case 2:      numer = 1; denom = 4; incr = 60; break;
344     case 3:      numer = 1; denom = 2; incr = 30; break;
345     case 4:      numer = 3; denom = 4; incr = 60; break;
346     default:     numer = 0; denom = 1; incr = 30; break;
347     }
348     switch ( (testCase - LARGE_SIZES) / LARGE_SIZES_NUMBER_OF_TESTS ) {
349     case 1:

```

```

341         g = new mgtaylor.graph.graphTypes.TriangularPrismMaximalPlanarGraph(
342             1000 );
343         performCodeWarmup( g, CODE_WARMUP_REPETITIONS );
344         speedTestSimilarTriangleGraphsWithIncreasingSizes( System.out, 501,
345             14999, incr, REPETITIONS_LARGE, number, denom );
346         break;
347     case 2:
348         g = new mgtaylor.graph.graphTypes.OrderedFaceTrisectionGraph( 1000 );
349         performCodeWarmup( g, CODE_WARMUP_REPETITIONS );
350         speedTestSimilarOrderedGraphsWithIncreasingSizes( System.out, 501,
351             14999, incr, REPETITIONS_LARGE, number, denom );
352         break;
353     case 3:
354         final mgtaylor.graph.graphTypes.RandomMaximalPlanarGraphGenerator gg
355             = new
356                 mgtaylor.graph.graphTypes.RandomMaximalPlanarGraphGenerator(
357                     15000 );
358         g = gg.generateGraph(1000);
359         performCodeWarmup( g, CODE_WARMUP_REPETITIONS );
360         speedTestSimilarRandomGraphsWithIncreasingSizes( System.out, gg,
361             502, 15000, incr, REPETITIONS_LARGE, number, denom );
362         break;
363     case 0:
364     default:
365         g = new mgtaylor.graph.graphTypes.PlanarMultiWheelGraph( 1000, 3,
366             true );
367         performCodeWarmup( g, CODE_WARMUP_REPETITIONS );
368         speedTestSimilarWheelGraphsWithIncreasingSizes( System.out, 502,
369             15000, incr, REPETITIONS_LARGE, number, denom );
370         break;
371     }
372 } else if ( EDGE_SIZES <= testCase && testCase < (EDGE_SIZES + 4 *
373     EDGE_SIZES_NUMBER_OF_TESTS) ) {
374     final mgtaylor.graph.graphTypes.ReduceableBiconnectedGraphGenerator gg;
375     final Graph g;
376     final int size, number, denom, incr;
377     switch ( (testCase - EDGE_SIZES) % EDGE_SIZES_NUMBER_OF_TESTS ) {
378     case 0:      number = 0; denom = 1; incr = 30; break;
379     case 1:      number = 1; denom = 1; incr = 30; break;
380     case 2:      number = 1; denom = 4; incr = 60; break;
381     case 3:      number = 1; denom = 2; incr = 30; break;
382     case 4:      number = 3; denom = 4; incr = 60; break;
383     default:     number = 0; denom = 1; incr = 30; break;
384     }
385     switch ( (testCase - EDGE_SIZES) / EDGE_SIZES_NUMBER_OF_TESTS ) {
386     case 1:
387         size = 14999;
388         g = new mgtaylor.graph.graphTypes.TriangularPrismMaximalPlanarGraph(
389             size );
390         break;
391     case 2:
392         size = 14999;
393         g = new mgtaylor.graph.graphTypes.OrderedFaceTrisectionGraph( size );
394         break;
395     case 3:
396         size = 14999;
397         g = new mgtaylor.graph.graphTypes.RandomMaximalPlanarGraphGenerator(
398             size ).generateGraph(size);
399         break;
400     case 0:
401     default:
402         size = 15000;
403         g = new mgtaylor.graph.graphTypes.PlanarMultiWheelGraph( size, 3,
404             true );
405         break;

```

```

394     }
395     gg = new mgtaylor.graph.graphTypes.ReduceableBiconnectedGraphGenerator(
        g, g.getVertices().get( getRootIndex( g.vertexSize(), numer, denom ) )
    );
396     speedTestGraphWithReducingEdges( System.out, gg, incr,
        REPETITIONS_LARGE );
397 }
398 }

400 public static void main( final String[] args ) {
401     if ( args.length > 0 ) {
402         for ( int i = 0; i < args.length; i++ ) {
403             String testCase = args[i];
404             if ( testCase.equalsIgnoreCase( "--Roots" ) ) {
405                 String file = args[++i];
406                 int graph = Integer.parseInt( args[++i] );
407                 java.util.ArrayList<String> graphs =
                    GraphExtractorLogParser.parseLogFile( file );
408                 final Graph g = new Graph( graphs.get( graph ) );
409                 file = null;
410                 graphs = null;
411                 performCodeWarmup( g, CODE_WARMUP_REPETITIONS );
412                 speedTestGraphWithDifferentRoot( System.out, g, REPETITIONS_ROOTS
                    );
413             } else {
414                 performTestCase( Integer.parseInt( testCase ) );
415                 System.gc();
416             }
417         }
418     } else
419         performTestCase( 0 );

421     System.out.close();
422 }
423 }

```

D.4.2 mgtaylor.graph.test.parsers.GraphExtractorLogParser

```

1 package mgtaylor.test.parsers;

3 import java.io.BufferedReader;
4 import java.io.File;
5 import java.io.FileReader;

7 public class GraphExtractorLogParser {

9     public static java.util.ArrayList<String> parseLogFile( final String filename
10 ) {
11         final java.util.ArrayList<String> graphs = new
            java.util.ArrayList<String>();
12         final File in = new File( filename );
13         try {
14             final BufferedReader br
15                 = new BufferedReader( new FileReader(
16                     in ) );

17             String string = br.readLine();
18             while ( string != null ) {
19                 if ( string.startsWith( "nodes { " ) ) {
20                     final StringBuffer graphString = new StringBuffer( string );
21                     while ( !string.equals( "}" ) ) {
22                         string = br.readLine();
23                         graphString.append( '\n' );

```

```

22         graphString.append( string );
23     }
24     graphs.add( graphString.toString() );
25 }
26 string = br.readLine();
27 }
28 br.close();
29 } catch ( Exception e ) {
30 }
31 return graphs;
32 }

34 /**
35  * @param args
36  */
37 public static void main( final String[] args ) {
38     // TODO Auto-generated method stub
39     for ( String arg : args ) {
40         java.util.ArrayList<String> graphs = parseLogFile( arg );
41         for ( String graph : graphs ) {
42             System.out.println( graph );
43             System.out.println();
44         }
45     }
46 }
48 }

```

D.4.3 mgtaylor.graph.test.parsers.GraphStatsLogParser

```

1 package mgtaylor.test.parsers;

3 import java.io.BufferedReader;
4 import java.io.BufferedWriter;
5 import java.io.File;
6 import java.io.FileReader;
7 import java.io.FileWriter;
8 import java.io.PrintWriter;

10 import mgtaylor.graph.Graph;

12 public class GraphStatsLogParser {
13     // "[GC [DefNew: 101476K->4947K(108864K), 0.0323195 secs]
14     // 272464K->175936K(1560768K), 0.0323640 secs] [Times: user=0.03 sys=0.00,
15     // real=0.03 secs] "

16     public static final boolean DISPLAY_NUMBER_OF_VERTICES = false;
17     public static final boolean DISPLAY_NUMBER_OF_EDGES = false;
18     public static final boolean DISPLAY_NUMBER_OF_SEGMENTS = true;

19     public static String extendFilename( final String filename, final String
20     extension, final String appendString ) {
21         return filename.replaceAll( extension + "$", appendString + extension );
22     }

23     public static void parseLogFile( final String filename ) {
24         File in = new File( filename );
25         File out = new File( extendFilename( filename, ".log", "_Depths" ) );

27         if ( !in.exists() || out.exists() ) {
28             System.err.println( "Check whether input file exists or output files do
29             not exist." );

```

```

29         return;
30     }
31     final PrintWriter writer;

32
33     try {
34         writer = new PrintWriter( new BufferedWriter( new FileWriter( out ) ) );
35     } catch ( Exception e ) {
36         e.printStackTrace();
37         return;
38     }

39
40     try {
41         final BufferedReader br          = new BufferedReader( new FileReader(
42             in ) );
43         Graph g = null;

44         String string = br.readLine();
45         long vertices = 0;
46         long edges = 0;
47         int root = -100;
48         while ( string != null ) {
49             if ( string.startsWith( "nodes { " ) ) {
50                 String graphString = string;
51                 while ( !string.equals( "}" ) ) {
52                     string = br.readLine();
53                     graphString = graphString + "\n" + string;
54                 }
55                 g = new Graph( graphString );
56                 vertices = g.vertexSize();
57                 edges = g.edgeSize();
58                 string = br.readLine();
59                 System.out.println( vertices + "\t" + edges );
60                 root = -1;
61                 continue;
62             } else if ( string.startsWith( "Root: " ) ) {
63                 String s = string.substring(6);
64                 int l = s.indexOf( '[' );
65                 if ( l > 0 ) {
66                     s = s.substring( 0, l );
67                     System.out.println( s + " : " + string );
68                 }
69                 int r = Integer.parseInt( s );
70                 if ( r != root && g != null ) {
71                     root = r;
72                     g.DFS( g.getVertices().get( r - 1 ) );
73                     g.generateSegments();
74                     if ( DISPLAY_NUMBER_OF_VERTICES )
75                         writer.print( g.vertexSize() + "\t" );
76                     if ( DISPLAY_NUMBER_OF_EDGES )
77                         writer.print( g.edgeSize() + "\t" );
78                     writer.print( r + "\t" + g.calculateAverageSegmentDepth() );
79                     if ( DISPLAY_NUMBER_OF_SEGMENTS )
80                         writer.print( "\t" + g.getNumberOfSegments() );
81                     writer.println();
82                 }
83             }
84             string = br.readLine();
85         }
86         br.close();
87         writer.close();
88     } catch ( Exception e ) {
89         try {
90             writer.close();
91         } catch ( Exception ex ) {}
92         e.printStackTrace();

```



```

93     }
94 }

96 public static final void main( String[] args ) {
97     for ( final String filename: args )
98         parseLogFile( filename );
99 }
100 }

```

D.4.4 mgtaylor.graph.test.parsers.NewGCLogParser

```

1 package mgtaylor.test.parsers;

3 import java.io.BufferedReader;
4 import java.io.BufferedWriter;
5 import java.io.File;
6 import java.io.FileReader;
7 import java.io.FileWriter;
8 import java.io.PrintWriter;

10 import java.util.regex.Matcher;
11 import java.util.regex.Pattern;

13 import mgtaylor.graph.Graph;

15 public class NewGCLogParser {
16     private static final String GC_STRING =
17         "\\[.*(\\d+\\.\\d+)\\s+secs\\]\\s+\\[Times:.+\\]\\s*";
18     public static final Pattern GC_PATTERN = Pattern.compile( "^" +
19         GC_STRING + "$" );
20     public static final Pattern ROOT_PATTERN = Pattern.compile( "^" +
21         Graph.ROOT_STRING + ": ([^\\[]*)( " + GC_STRING + ")?$" );
22     public static final Pattern START_PATTERN = Pattern.compile( "^" +
23         Graph.START_STRING + ".*$" );
24     public static final Pattern DFS_PATTERN = Pattern.compile( "^" +
25         Graph.DFS_STRING + ": (\\d+).*$" , Pattern.CASE_INSENSITIVE );
26     public static final Pattern GENERATION_PATTERN = Pattern.compile( "^" +
27         Graph.GENERATION_STRING + ": (\\d+).*$" );
28     public static final Pattern EMBEDDING_PATTERN = Pattern.compile( "^" +
29         Graph.EMBEDDING_STRING + ": (\\d+).*$" );
30     public static final Pattern EDGE_SIDES_PATTERN = Pattern.compile( "^" +
31         Graph.EDGE_SIDES_STRING + ": (\\d+).*$" );

33     private static final boolean OUTPUT_TOTAL = true;
34     private static final boolean OUTPUT_DFS = false;
35     private static final boolean OUTPUT_GENERATION = false;
36     private static final boolean OUTPUT_EMBEDDING = false;
37     private static final boolean OUTPUT_NON_EMBEDDING = false;
38     private static final boolean OUTPUT_EDGES = true;
39     private static final boolean OUTPUT_DEPTHS = true;

41     private static final boolean DEPTHS_DISPLAY_NUMBER_OF_VERTICES = false;
42     private static final boolean DEPTHS_DISPLAY_NUMBER_OF_EDGES = false;
43     private static final boolean DEPTHS_DISPLAY_NUMBER_OF_SEGMENTS = true;

45     public static String extendFilename( final String filename, final String
46         extension, final String appendString ) {
47         return filename.replaceAll( extension + "$", appendString + extension );
48     }

49     @SuppressWarnings( "unused" )
50     public static void parseLogFile( final String filename ) {

```

```

43     final File in = new File( filename );
44     final File out      = new File( extendFilename( filename, ".log",
45         "_Total" ) );
46     final File out_DFS  = new File( extendFilename( filename, ".log",
47         "_DFS" ) );
48     final File out_Generation = new File( extendFilename( filename, ".log",
49         "_Generation" ) );
50     final File out_Embedding  = new File( extendFilename( filename, ".log",
51         "_Embedding" ) );
52     final File out_NonEmbedding = new File( extendFilename( filename,
53         ".log", "_NonEmbedding" ) );
54     final File out_Edges      = new File( extendFilename( filename, ".log",
55         "_EdgeSides" ) );
56     final File out_Depths     = new File( extendFilename( filename, ".log",
57         "_Depths" ) );
58
59     if ( in.exists() &&
60         (!OUTPUT_TOTAL      || !out.exists()) &&
61         (!OUTPUT_DFS        || !out_DFS.exists()) &&
62         (!OUTPUT_GENERATION || !out_Generation.exists()) &&
63         (!OUTPUT_EMBEDDING  || !out_Embedding.exists()) &&
64         (!OUTPUT_NON_EMBEDDING || !out_NonEmbedding.exists()) &&
65         (!OUTPUT_EDGES      || !out_Edges.exists()) &&
66         (!OUTPUT_DEPTHS     || !out_Depths.exists())
67     ) {
68     } else {
69         System.err.println( "Check whether input file exists or output files do
70             not exist." );
71         return;
72     }
73
74     final PrintWriter writer;
75     final PrintWriter writer_DFS;
76     final PrintWriter writer_Generation;
77     final PrintWriter writer_Embedding;
78     final PrintWriter writer_NonEmbedding;
79     final PrintWriter writer_Edges;
80     final PrintWriter writer_Depths;
81
82     try {
83         if ( OUTPUT_TOTAL )      writer = new PrintWriter( new
84             BufferedWriter( new FileWriter( out ) ) );
85         else                    writer = null;
86         if ( OUTPUT_DFS )      writer_DFS = new PrintWriter( new
87             BufferedWriter( new FileWriter( out_DFS ) ) );
88         else                    writer_DFS = null;
89         if ( OUTPUT_GENERATION ) writer_Generation = new PrintWriter( new
90             BufferedWriter( new FileWriter( out_Generation ) ) );
91         else                    writer_Generation = null;
92         if ( OUTPUT_EMBEDDING ) writer_Embedding = new PrintWriter( new
93             BufferedWriter( new FileWriter( out_Embedding ) ) );
94         else                    writer_Embedding = null;
95         if ( OUTPUT_NON_EMBEDDING ) writer_NonEmbedding = new PrintWriter(
96             new BufferedWriter( new FileWriter( out_NonEmbedding ) ) );
97         else                    writer_NonEmbedding = null;
98         if ( OUTPUT_EDGES )      writer_Edges = new PrintWriter( new
99             BufferedWriter( new FileWriter( out_Edges ) ) );
100        else                    writer_Edges = null;
101        if ( OUTPUT_DEPTHS )      writer_Depths = new PrintWriter( new
102            BufferedWriter( new FileWriter( out_Depths ) ) );
103        else                    writer_Depths = null;
104    } catch ( Exception e ) {
105        e.printStackTrace();
106        return;
107    }

```

```

93     try {
94         final BufferedReader br          = new BufferedReader( new FileReader(
           in ) );

96         String string = br.readLine();
97         long vertices = 0;
98         long edges = 0;
99         Graph g = null;
100         int r = -1;
101         while ( string != null ) {
102             if ( string.startsWith( "nodes { " ) ) {
103                 String graphString = string;
104                 while ( !string.equals( "}" ) ) {
105                     string = br.readLine();
106                     graphString = graphString + "\n" + string;
107                 }
108                 g = new Graph( graphString );
109                 vertices = g.vertexSize();
110                 edges = g.edgeSize();
111                 r = -2;
112                 string = br.readLine();
113                 System.out.println( vertices + "\t" + edges );
114                 continue;
115             }
116             Matcher m_root = ROOT_PATTERN.matcher( string );
117             if ( m_root.matches() ) {
118                 String root = m_root.group(1).replaceFirst( "~\\D+", "" );
119                 if ( OUTPUT_DEPTHS && ( r == -2 || r != Integer.parseInt( root ) ) ) {
120                     r = Integer.parseInt( root );
121                     g.DFS( g.getVertices().get( r - 1 ) );
122                     g.generateSegments();
123                     if ( DEPTHS_DISPLAY_NUMBER_OF_VERTICES )
124                         writer_Depths.print( g.vertexSize() + "\t" );
125                     if ( DEPTHS_DISPLAY_NUMBER_OF_EDGES )
126                         writer_Depths.print( g.edgeSize() + "\t" );
127                     writer_Depths.print( r + "\t" +
128                         g.calculateAverageSegmentDepth() );
129                     if ( DEPTHS_DISPLAY_NUMBER_OF_SEGMENTS ) {
130                         writer_Depths.print( "\t" + g.getNumberOfSegments() );
131                         g.embedSegments();
132                         writer_Depths.print( "\t" + g.calculateTotalConflicts() );
133                         writer_Depths.print( "\t" + g.calculateNumberOfBlocks() );
134                         writer_Depths.print( "\t" + g.edgeSize() );
135                         writer_Depths.print( "\t" + g.calculateNumberOfFlippableBlocks() );
136                         writer_Depths.print( "\t" + g.getPermutations().size() );
137                     }
138                     writer_Depths.println();
139                 }
140                 int gc_count = 0;
141                 Matcher m_start = START_PATTERN.matcher( string = br.readLine() );
142                 while ( !m_start.matches() )
143                     m_start = START_PATTERN.matcher( string = br.readLine() );
144                 Matcher m_dfs = DFS_PATTERN.matcher( string = br.readLine() );
145                 long gc_time = 0;
146                 int gc_temp_count = 0;
147                 while ( !m_dfs.matches() ) {
148                     Matcher gc = GC_PATTERN.matcher( string );
149                     if ( gc.matches() ) {
150                         gc_time += Math.round( Double.parseDouble( gc.group(1) ) *
151                             1000000000 );
152                         gc_temp_count++;
153                         gc_count++;
154                     } else {

```

```

153         if ( string.compareTo( "" ) != 0 )
154             System.err.println( string );
155     }
156     m_dfs = DFS_PATTERN.matcher( string = br.readLine() );
157 }
158 long dfs_time = Long.parseLong( m_dfs.group(1) ) - gc_time;
159 if (OUTPUT_DFS) writer_DFS.println( vertices + "\t" + edges +
160     "\t" + root + "\t" + dfs_time + "\t" + gc_temp_count );
161 Matcher m_generation = GENERATION_PATTERN.matcher( string =
162     br.readLine() );
163 gc_time = 0;
164 gc_temp_count = 0;
165 while (!m_generation.matches() ) {
166     Matcher gc = GC_PATTERN.matcher( string );
167     if ( gc.matches() ) {
168         gc_time += Math.round( Double.parseDouble( gc.group(1) ) *
169             1000000000 );
170         gc_temp_count++;
171         gc_count++;
172     } else {
173         if ( string.compareTo( "" ) != 0 )
174             System.err.println( string );
175     }
176     m_generation = GENERATION_PATTERN.matcher( string =
177         br.readLine() );
178 }
179 long generation_time = Long.parseLong( m_generation.group(1) ) -
180 gc_time;
181 if (OUTPUT_GENERATION) writer_Generation.println( vertices + "\t"
182 + edges + "\t" + root + "\t" + generation_time + "\t" +
183 gc_temp_count );
184 Matcher m_embedding = EMBEDDING_PATTERN.matcher( string =
185     br.readLine() );
186 gc_time = 0;
187 gc_temp_count = 0;
188 while (!m_embedding.matches() ) {
189     Matcher gc = GC_PATTERN.matcher( string );
190     if ( gc.matches() ) {
191         gc_time += Math.round( Double.parseDouble( gc.group(1) ) *
192             1000000000 );
193         gc_temp_count++;
194         gc_count++;
195     } else {
196         if ( string.compareTo( "" ) != 0 )
197             System.err.println( string );
198     }
199     m_embedding = EMBEDDING_PATTERN.matcher( string =
200         br.readLine() );
201 }
202 final int gc_nonembed_count = gc_count - gc_temp_count;
203 long embedding_time = Long.parseLong( m_embedding.group(1) ) -
204 gc_time;
205 if (OUTPUT_EMBEDDING) writer_Embedding.println( vertices + "\t" +
206     edges + "\t" + root + "\t" + embedding_time + "\t" +
207     gc_temp_count );
208 Matcher m_edge_sides = EDGE_SIDES_PATTERN.matcher( string =
209     br.readLine() );
210 gc_time = 0;
211 gc_temp_count = 0;
212 while (!m_edge_sides.matches() ) {
213     Matcher gc = GC_PATTERN.matcher( string );
214     if ( gc.matches() ) {
215         gc_time += Math.round( Double.parseDouble( gc.group(1) ) *
216             1000000000 );
217         gc_temp_count++;

```

```

203         gc_count++;
204     } else {
205         if ( string.compareTo( "" ) != 0 )
206             System.err.println( string );
207     }
208     m_edge_sides = EDGE_SIDES_PATTERN.matcher( string =
        br.readLine() );
209 }
210 long edge_sides_time = Long.parseLong( m_edge_sides.group(1) ) -
    gc_time;
211 if (OUTPUT_EDGES) writer_Edges.println( vertices + "\t" + edges +
    "\t" + root + "\t" + edge_sides_time + "\t" + gc_temp_count );
212 long total_time = dfs_time + generation_time + edge_sides_time;
213 if (OUTPUT_NON_EMBEDDING) writer_NonEmbedding.println( vertices +
    "\t" + edges + "\t" + root + "\t" + total_time + "\t" +
    (gc_nonembed_count + gc_temp_count) );
214 total_time += embedding_time;
215 if (OUTPUT_TOTAL) writer.println( vertices + "\t" + edges + "\t"
    + root + "\t" + total_time + "\t" + gc_count );
216 }
217 string = br.readLine();
218 }
219 br.close();
220 if (OUTPUT_TOTAL)        writer.close();
221 if (OUTPUT_DFS)          writer_DFS.close();
222 if (OUTPUT_GENERATION)   writer_Generation.close();
223 if (OUTPUT_EMBEDDING)    writer_Embedding.close();
224 if (OUTPUT_NON_EMBEDDING) writer_NonEmbedding.close();
225 if (OUTPUT_EDGES)        writer_Edges.close();
226 if (OUTPUT_DEPTHS)       writer_Depths.close();
227 } catch ( Exception e ) {
228     try {
229         if (OUTPUT_TOTAL)        writer.close();
230         if (OUTPUT_DFS)          writer_DFS.close();
231         if (OUTPUT_GENERATION)   writer_Generation.close();
232         if (OUTPUT_EMBEDDING)    writer_Embedding.close();
233         if (OUTPUT_NON_EMBEDDING) writer_NonEmbedding.close();
234         if (OUTPUT_EDGES)        writer_Edges.close();
235         if (OUTPUT_DEPTHS)       writer_Depths.close();
236     } catch ( Exception ex ) {}
237     e.printStackTrace();
238 }
239 }

241 public static final void main( String[] args ) {
242     for ( final String filename: args )
243         parseLogFile( filename );
244 }
245 }

```

D.4.5 mgtaylor.graph.test.utilities.ConcatenateFiles

```

1 package mgtaylor.test.utilities;

3 import java.io.*;

5 public class ConcatenateFiles {

7     /**
8      * @param args
9      */
10    public static void main(String[] args) {

```

```

11     if ( args.length < 3 ) {
12         System.out.println( "Usage: java ConcatenateFiles file1 file2 [...
            fileN] outputFile" );
13         System.exit( 1 );
14     }
15     File out = new File( args[args.length - 1] );
16     try {
17         PrintWriter writer = new PrintWriter( new BufferedWriter( new
            FileWriter( out, out.exists() ) ) );
18         for ( int i = 0; i < args.length - 1; i++ ) {
19             File in = new File( args[i] );
20             if ( in.exists() ) {
21                 BufferedReader br = new BufferedReader( new FileReader( in ) );
22                 String s = null;
23                 while ( ( s = br.readLine() ) != null )
24                     writer.println( s );
25                 br.close();
26             }
27         }
28         writer.close();
29     } catch ( IOException e ) {
30         e.printStackTrace();
31     }
32 }
34 }

```

D.4.6 mgtaylor.graph.test.utilities.SplitLogFiles

```

1 package mgtaylor.test.utilities;

3 import java.io.BufferedReader;
4 import java.io.BufferedWriter;
5 import java.io.File;
6 import java.io.FileReader;
7 import java.io.FileWriter;
8 import java.io.PrintWriter;
9 import mgtaylor.test.PlanaritySpeedTestViewer;

11 public class SplitLogFile {
12     private static enum CompareCol { VERTEX, EDGE, ROOT, DEPTHS };
13     private static enum CompareType { ASC, EQ, DESC };

15     private static CompareCol getColumn( final String arg ) {
16         final String[] in = { "vertex", "edge", "root", "depths" };
17         final CompareCol[] out = { CompareCol.VERTEX, CompareCol.EDGE,
            CompareCol.ROOT, CompareCol.DEPTHS };
18         for ( int i = 0; i < in.length; i++ )
19             if ( in[i].compareToIgnoreCase( arg ) == 0 ) return out[i];
20         throw new IllegalArgumentException( "Invalid Column Type" );
21     }

23     private static int getColumnIndex( final CompareCol col ) {
24         if ( col == CompareCol.EDGE ) return 1;
25         if ( col == CompareCol.ROOT ) return 2;
26         return 0;
27     }

29     private static String getColumnPostfix( final CompareCol col ) {
30         if ( col == CompareCol.DEPTHS )
31             return "Depths";
32         return "Total";

```

```

33     }

35     private static CompareType getCompareType( final String arg ) {
36         final String[] in      = { "ASC", "EQ", "DESC" };
37         final CompareType[] out = { CompareType.ASC, CompareType.EQ,
38             CompareType.DESC };
39         for ( int i = 0; i < in.length; i++ )
40             if ( in[i].compareToIgnoreCase( arg ) == 0 ) return out[i];
41         throw new IllegalArgumentException( "Invalid Comparison Type" );
42     }

43     private static boolean compareAgainstType( final int prev, final int curr,
44         final CompareType type ) {
45         if ( prev == -1 ) return true;
46         if ( type == CompareType.DESC )
47             return prev >= curr;
48         if ( type == CompareType.ASC )
49             return prev <= curr;
50         return prev == curr;
51     }

52     private static int UnknownOutputIndex = 1;

54     private static String getOutputName( String[] cols, CompareCol col,
55         CompareType comp ) {
56         if ( comp == CompareType.EQ )
57             return cols[ getColumnIndex( col ) ];
58         final int root = Integer.parseInt( cols[ getColumnIndex( CompareCol.ROOT )
59             ] );
60         final int vert = Integer.parseInt( cols[ getColumnIndex( CompareCol.VERTEX
61             ) ] );
62         if ( root == PlanaritySpeedTestViewer.getRootIndex( vert, 0, 1 ) )
63             return "0";
64         if ( root == PlanaritySpeedTestViewer.getRootIndex( vert, 1, 1 ) )
65             return "N";
66         if ( root == PlanaritySpeedTestViewer.getRootIndex( vert, 1, 4 ) )
67             return "N4";
68         if ( root == PlanaritySpeedTestViewer.getRootIndex( vert, 1, 2 ) )
69             return "N2";
70         if ( root == PlanaritySpeedTestViewer.getRootIndex( vert, 3, 4 ) )
71             return "3N4";
72         return "U" + (UnknownOutputIndex++);
73     }

74     private static final String USAGE =
75         "Usage:\n" +
76         "    SplitLogFile <FILENAME> <SPLITCOLUMN> <SPLITCOMPARISON>
77         <OUTPUT_PREFIX>\n" +
78         "\n" +
79         "        FILENAME      : The path of the input file.\n" +
80         "        SPLITCOLUMN    := [ VERTEX | EDGE | ROOT | DEPTHS ]\n" +
81         "        SPLITCOMPARISON := [ ASC | EQ | DESC ]\n" +
82         "        OUTPUTPREFIX   : The prefix for the output files.\n";

83     public static void main( final String[] args ) {
84         if ( args.length != 4 ) {
85             System.out.println( USAGE );
86             return;
87         }

88         final String filename      = args[0];
89         final CompareCol col;
90         final int column;
91         final String postfix;
92         final CompareType compare;

```

```

87     final String outputname    = args[3];

89     try {
90         col        = getColumn( args[1] );
91         column     = getColumnIndex( col );
92         postfix    = getColumnPostfix( col );
93         compare    = getCompareType( args[2] );
94     } catch (IllegalArgumentException e) {
95         System.out.println( USAGE );
96         return;
97     }

99     File in = new File( filename );
100    if ( !in.exists() ) {
101        System.err.println( "Check whether input file exists." );
102        return;
103    }

105    PrintWriter writer = null;
106    try {
107        final BufferedReader br = new BufferedReader( new FileReader( in ) );

109        int curr = -1;
110        String string = br.readLine();
111        File out;

113        while (string != null) {
114            String[] cols = string.split( "\\t" );
115            final int prev = curr;
116            curr = Integer.parseInt( cols[column] );
117            if ( compareAgainstType( prev, curr, compare ) ) {
118                out = new File( outputname + "-" + getOutputName( cols, col,
119                    compare ) + "-" + postfix + ".log" );
120                try {
121                    if ( writer != null )
122                        writer.close();
123                    writer = new PrintWriter( new BufferedWriter( new FileWriter(
124                        out ) ) );
125                } catch ( Exception e ) {
126                    e.printStackTrace();
127                    br.close();
128                    if ( writer != null )
129                        writer.close();
130                    return;
131                }
132            }
133            writer.println( string );
134            string = br.readLine();
135        }
136        br.close();
137        if ( writer != null )
138            writer.close();
139    } catch ( Exception e ) {
140        try {
141            writer.close();
142        } catch ( Exception ex ) {}
143        e.printStackTrace();
144    }
145 }

```


D.5 GUI Files

D.5.1 mgtaylor.display.AngleHighlightView

```

1 package mgtaylor.display;

3 import java.awt.*;
4 import java.awt.geom.*;
5 import java.util.*;

7 import mgtaylor.layout.*;
8 import mgtaylor.layout.event.*;
9 import mgtaylor.planarity.event.*;

11 import mgtaylor.graph.*;

13 public class AngleHighlightView extends DraggableView implements
PlanarOrderListener {
14     /**/
15     private static final long serialVersionUID = -1624845479870365270L;
16     private static final int ORDERED_INCORRECTLY = 0;
17     private static final int ORDERED_CLOCKWISE = 1;
18     private static final int ORDERED_ANTICLOCKWISE = 2;

20     private final PlanarityHandler planarLayout;

22     private final HashMap<Edge, Double> edgeAngle = new HashMap<Edge,
Double>();
23     private final HashMap<Vertex, Edge[]> optimalOrder = new HashMap<Vertex,
Edge[]>();
24     private final HashMap<Vertex, Edge[]> currentOrder = new HashMap<Vertex,
Edge[]>();
25     private final HashMap<Vertex, Boolean> correctOrder = new HashMap<Vertex,
Boolean>();

27     protected Double orderVertexBorder = 3.0d;

29     private Color clockwiseOrderColour = Color.yellow;
30     private Color anticlockwiseOrderColour = new Color(96, 96, 255);
31     private Color correctOrderColour = Color.green;
32     private Color errorOrderColour = Color.red;
33     private BasicStroke orderStroke = new BasicStroke( 1.0f );

35     private Color nonPlanarSubGraphEdgeColour = Color.red;
36     private Color nonPlanarOtherEdgeColour = new Color( 0, 0, 0, 196 );
37     private BasicStroke nonPlanarSubGraphEdgeStroke = new BasicStroke( 2.0f );
38     private BasicStroke nonPlanarOtherEdgeStroke = new BasicStroke( 1.0f );

40     public Color getClockwiseOrderColour() { return clockwiseOrderColour; }
41     public Color getAnticlockwiseOrderColour() { return
anticlockwiseOrderColour; }
42     public Color getErrorOrderColour() { return errorOrderColour; }
43     public BasicStroke getOrderStroke() { return orderStroke; }

45     public AngleHighlightView(final VertexLayout layout, final PlanarityHandler
planarity) {
46         super(layout);
47         planarLayout = planarity;
48         generateInitialData();
49         planarity.addPlanarOrderListener( this );
50     }

52     @Override

```

```

53     protected void generateInitialData() {
54         if ( getGraphLayout().getGraph().isPlanar() ) {
55             for (Edge edge: getGraphLayout().getGraph().getEdges())
56                 edgeAngle.put(edge, generateEdgeAngle(edge));
57             for ( Vertex vertex: getGraphLayout().getGraph().getVertices() ) {
58                 optimalOrder.put( vertex, vertex.getEdgeOrder() );
59                 currentOrder.put( vertex, generateCurrentEdgeOrder(vertex) );
60             // System.out.println( vertex.toString() + ": " +
61             optimalOrder.get(vertex).toString() + " -> " + arrayToString( currentOrder.get(
62                 vertex ) ));
63             }
64         }
65     }
66
67     protected double getEdgeAngle( Edge edge, Vertex vertex ) {
68         if ( edge.getFrom() == vertex ) {
69             return edgeAngle.get( edge );
70         } else if ( edge.getTo() == vertex ) {
71             Double angle = edgeAngle.get( edge );
72             if ( angle != null ) {
73                 if ( angle < 180 )
74                     return angle + 180;
75                 return angle - 180;
76             }
77         }
78         return Double.NaN;
79     }
80
81     protected double generateEdgeAngle( Edge edge ) {
82         Point2D.Double from = getGraphLayout().getVertexLocation( edge.getFrom() );
83         Point2D.Double to = getGraphLayout().getVertexLocation( edge.getTo() );
84         double x = to.x - from.x;
85         double y = to.y - from.y;
86         double angle = Math.toDegrees( Math.atan2(-y, x) );
87         if ( angle < 0 )
88             angle += 360;
89         return angle;
90     }
91
92     @Override
93     protected void paintSetVertexFillStyle( Vertex vertex, Graphics2D g2d ) {
94         super.paintSetVertexFillStyle(vertex, g2d);
95         if ( getGraphLayout().getGraph().isPlanar() && correctOrder.get( vertex ) )
96             g2d.setColor( correctOrderColour );
97     }
98
99     @Override
100     protected void paintSetEdgeStyle( Edge edge, Graphics2D g2d ) {
101         if ( planarLayout.isNonPlanarEdge(edge) ) {
102             g2d.setColor( nonPlanarSubGraphEdgeColour );
103             g2d.setStroke( nonPlanarSubGraphEdgeStroke );
104         } else {
105             g2d.setColor( nonPlanarOtherEdgeColour );
106             g2d.setStroke( nonPlanarOtherEdgeStroke );
107         }
108     }
109
110     @Override
111     public void paint( Graphics g, boolean paintBackground ) {
112         if ( bounds == null )
113             return;
114         Graphics2D g2d = (Graphics2D) g;
115         if ( paintBackground )
116             beforePaint( g2d );

```

```

115         if ( getGraphLayout().getGraph().isPlanar() ) {
116             g2d.setStroke( getOrderStroke() );
117             for ( Vertex vertex : getGraphLayout().getGraph().getVertices() ) {
118                 if ( vertex.size() > 2 ) {
119                     Edge[] current = currentOrder.get( vertex );
120                     int ordering = ORDERED_CLOCKWISE + ORDERED_ANTICLOCKWISE;
121                     for ( int i = 0; i < current.length; i++ ) {
122                         Edge e1 = current[ i ];
123                         Edge e2 = current[ (i + 1) % current.length ];
124                         double a1 = getEdgeAngle( e1, vertex );
125                         double a2 = getEdgeAngle( e2, vertex );
126                         double e = a2 - a1;
127                         if ( e < 0 )
128                             e += 360;
129                         final Shape s = this.getVertexShape( vertex );
130                         Rectangle b = s.getBounds();
131                         Shape shape = new Arc2D.Double(
132                             b.x - orderVertexBorder,
133                             b.y - orderVertexBorder,
134                             b.width + 2 * orderVertexBorder,
135                             b.height + 2 * orderVertexBorder,
136                             a1,
137                             e,
138                             Arc2D.PIE
139                         );
140                         if ( e2 == vertex.getClockwiseFrom( e1 ) ) {
141                             g2d.setColor( getClockwiseOrderColour() );
142                             ordering &= ORDERED_CLOCKWISE;
143                         } else if ( e2 == vertex.getAntiClockwiseFrom( e1 ) ) {
144                             g2d.setColor( getAnticlockwiseOrderColour() );
145                             ordering &= ORDERED_ANTICLOCKWISE;
146                         } else {
147                             g2d.setColor( getErrorOrderColour() );
148                             ordering &= ORDERED_INCORRECTLY;
149                         }
150                         g2d.fill( shape );
151                         g2d.draw( shape );
152                     }
153                     correctOrder.put( vertex, ordering != ORDERED_INCORRECTLY );
154                 } else {
155                     correctOrder.put( vertex, true );
156                 }
157             }
158         }
159         super.paint( g, false );
160         if ( paintBackground )
161             afterPaint( g2d );
162     }

164     protected Edge[] generateCurrentEdgeOrder( final Vertex vertex ) {
165         ArrayList<Edge> connectedEdges = vertex.getConnections();
166         Edge[] edges = connectedEdges.toArray( new Edge[ connectedEdges.size() ] );
167         Arrays.sort( edges, new Comparator<Edge>() {
168             @Override
169             public int compare( final Edge e1, final Edge e2 ) {
170                 double a1 = getEdgeAngle( e1, vertex );
171                 double a2 = getEdgeAngle( e2, vertex );
172                 if ( a1 < a2 ) return -1;
173                 if ( a1 > a2 ) return 1;
174                 return 0;
175             }
176         });
177         return edges;
178     }

```

```

180     @Override
181     public void handleGraphLayoutChanged(GraphLayoutEvent event) {
182         if ( getGraphLayout().getGraph().isPlanar() ) {
183             MovementMap map = event.getMovementMap();
184             final HashSet<Vertex> connected = new HashSet<Vertex>();
185             final HashSet<Edge> edges = new HashSet<Edge>();
186             for ( Vertex vertex: map.getVertices() ) {
187                 for ( Edge edge: vertex.getConnectedEdges() ) {
188                     connected.add( edge.getOtherEndFrom( vertex ) );
189                 }
190                 edges.addAll( vertex.getConnectedEdges() );
191             }
192             connected.addAll( map.getVertices() );
193             for (Edge edge: edges)
194                 edgeAngle.put(edge, generateEdgeAngle(edge));
195             for ( Vertex vertex: connected )
196                 currentOrder.put( vertex, generateCurrentEdgeOrder(vertex) );
197             // System.out.println();
198             // for ( Vertex vertex: getGraphLayout().getGraph().getVertices() )
199             // System.out.println( vertex.toString() + ": " +
200             // optimalOrder.get(vertex).toString() + " -> " + arrayToString( currentOrder.get(
201             // vertex ) ));
202             super.handleGraphLayoutChanged(event);
203         }
204
205         public static <E> String arrayToString(E[] array) {
206             StringBuffer buffer = new StringBuffer();
207             buffer.append('[');
208             for (int i = 0; i < array.length; i++) {
209                 if ( i > 0 )
210                     buffer.append( ", " );
211                 buffer.append( array[i].toString() );
212             }
213             buffer.append(']');
214             return buffer.toString();
215         }
216
217         @Override
218         public void handlePlanarOrderChanged( final PlanarOrderEvent event ) {
219             if ( planarLayout != event.getPlanarity() ) {
220                 System.err.println( "AngleHighlightView.handlePlanarOrderChanged() -
221                 incorrect planarity test.");
222                 return;
223             }
224             for ( Vertex vertex: planarLayout.getGraph().getVertices() )
225                 optimalOrder.put( vertex, event.getOrder( vertex ) );
226             repaint();
227         }
228     }

```

D.5.2 mgtaylor.display.Arrow

```

1 package mgtaylor.display;
2
3 import java.awt.geom.Point2D;
4 import java.awt.geom.Line2D.Double;
5
6 public class Arrow extends Double {
7
8     /**

```

```

9      *
10     */
11     private static final long serialVersionUID = 4441803674890709608L;

13     public Arrow(Point2D arg0, Point2D arg1) {
14         super(arg0, arg1);
15     }

17     public Arrow(double arg0, double arg1, double arg2, double arg3) {
18         super(arg0, arg1, arg2, arg3);
19     }
21 }

```

D.5.3 mgtaylor.display.DraggableView

```

1 package mgtaylor.display;

3 import java.awt.*;
4 import java.awt.event.*;
5 import java.awt.geom.*;
6 import java.util.*;

8 import mgtaylor.layout.*;

10 import mgtaylor.graph.*;

12 /**
13  * Draggable View is an extension of the basic View that allows
14  * vertices and edges to be moved from the initial positions by
15  * the user through mouse interactions.
16  *
17  * Clicking on a vertex/edge will deselect previous selections and
18  * select the clicked on vertex or edge and the edge's end vertices.
19  * Once selected the objects will be highlighted as well as any edges
20  * connected to the selected vertices. Dragging while clicked will
21  * create ghost copies of the selected objects in the location they
22  * have been dragged to, upon release of the mouse the selected
23  * objects will be moved by the offset dragged and a GraphDisplayEvent
24  * will fire.
25  *
26  * Clicking on an empty space and dragging will create a selection
27  * box and when the mouse is released the contents of the selection
28  * box will be selected.
29  *
30  * Holding Shift when clicking will cause the previous selections
31  * not to be deselected.
32  *
33  * Holding Control when clicking on a vertex/edge will toggle it's
34  * selection.
35  *
36  * @author Martyn G Taylor
37  */
38 /**
39  public abstract class DraggableView extends View {
40     /**/
41     private static final long serialVersionUID = 4531164691913408845L;

43     protected static final boolean SELECTED = true;
44     protected static final boolean DESELECTED = !SELECTED;

46     private Rectangle clickLocation = null;

```

```

47  private boolean clickedOnObject = false;
48  // private boolean objectStatus = SELECTED;
49  private boolean dragged = false;

51  private final HashSet<Edge> selectedEdges = new HashSet<Edge>();
52  private final HashSet<Vertex> selectedVertices = new HashSet<Vertex>();
53  private final HashSet<Edge> edgesConnectedToSelected = new HashSet<Edge>();

55  private BasicStroke selectedEdgeStroke = new BasicStroke(2.0f);
56  private BasicStroke selectedVertexStroke = new BasicStroke(2.0f);
57  private Color selectedEdgeColour = Color.black;
58  private Color selectedVertexColour = Color.black;
59  private Color selectedVertexFillColour = Color.white;
60  private Color selectedTextColour = Color.black;
61  private Font selectedLabelFont = new Font( getFont().getName(),
Font.BOLD, getFont().getSize() );

63  private BasicStroke connectedEdgeStroke = new BasicStroke(1.5f);
64  private Color connectedEdgeColour = Color.lightGray;

66  private Color dragColour = Color.blue;
67  private BasicStroke dragStroke = new BasicStroke(
68      1.0f, // Line width
69      BasicStroke.CAP_ROUND, // Line end cap style
70      BasicStroke.JOIN_ROUND, // Line join style
71      10.0f, // Miter Limit
72      new float[] { 5.0f, 5.0f }, // Dash Pattern
73      0.0f); // Dash phase

75  public DraggableView(VertexLayout layout) {
76      super(layout);
77  }

79  public DraggableView(View view) {
80      super(view);
81  }

83  public BasicStroke getSelectedEdgeStroke() { return
selectedEdgeStroke; }
84  public BasicStroke getSelectedVertexStroke() { return
selectedVertexStroke; }
85  public Color getSelectedEdgeColour() { return
selectedEdgeColour; }
86  public Color getSelectedVertexColour() { return
selectedVertexColour; }
87  public Color getSelectedVertexFillColour() { return
selectedVertexFillColour; }
88  public Color getSelectedTextColour() { return
selectedTextColour; }
89  public Font getSelectedFont() { return
selectedLabelFont; }

91  public BasicStroke getConnectedEdgeStroke() { return
connectedEdgeStroke; }
92  public Color getConnectedEdgeColour() { return
connectedEdgeColour; }

94  public void setSelectedEdgeStroke(BasicStroke stroke) {
selectedEdgeStroke = stroke; }
95  public void setSelectedVertexStroke(BasicStroke stroke) {
selectedVertexStroke = stroke; }
96  public void setSelectedEdgeColour(Color colour) {
selectedEdgeColour = colour; }
97  public void setSelectedVertexColour(Color colour) {
selectedVertexColour = colour; }

```

```

98     public void setSelectedVertexFillColour(Color colour)      {
99         selectedVertexFillColour = colour; }
100    public void setSelectedTextColour(Color colour)             {
101        selectedTextColour = colour; }
102    public void setSelectedFont(Font font)                       { selectedLabelFont
    = font; }
103    public void setSelectedFont(String name, int style, int size) {
    setSelectedFont(new Font( name, style, size )); }

104    public void clearSelected()                                  {
105        selectedVertices.clear(); selectedEdges.clear();
106        edgesConnectedToSelected.clear(); }
107    public boolean isSelected(Edge edge)                        { return
    selectedEdges.contains(edge); }
108    public boolean isConnected(Edge edge)                      { return
    edgesConnectedToSelected.contains(edge); }
109    public boolean isSelected(Vertex node)                     { return
    selectedVertices.contains(node); }

110    public void addSelectedVertices(Vertex vertex) {
111        if ( !isSelected( vertex ) ) {
112            selectedVertices.add(vertex);
113            for ( Edge edge: vertex.getConnectionEdges() ) {
114                if ( edgesConnectedToSelected.contains( edge ) ) {
115                    edgesConnectedToSelected.remove( edge );
116                    selectedEdges.add( edge );
117                } else {
118                    edgesConnectedToSelected.add( edge );
119                }
120            }
121        }
122    }
123    public void addSelectedVertices(Collection<Vertex> vertices) {
124        for ( Vertex vertex: vertices )
125            addSelectedVertices( vertex );
126    }

127    public void addSelectedEdges(Edge edge) {
128        addSelectedVertices( edge.getFrom() );
129        addSelectedVertices( edge.getTo() );
130    }

131    public void addSelectedEdges(Collection<Edge> edges) {
132        for ( Edge edge: edges )
133            addSelectedEdges( edge );
134    }

135    public void removeSelectedVertices(Vertex vertex) {
136        selectedVertices.remove( vertex );
137        for ( Edge edge: vertex.getConnectionEdges() ) {
138            if ( selectedEdges.contains( edge ) ) {
139                edgesConnectedToSelected.add( edge );
140                selectedEdges.remove( edge );
141            } else {
142                edgesConnectedToSelected.remove( edge );
143            }
144        }
145    }
146    }

147    public void removeSelectedVertices(Collection<Vertex> vertices) {
148        for ( Vertex vertex: vertices )
149            removeSelectedVertices( vertex );
150    }

151    public void removeSelectedEdges(Edge edge) {

```

```

154         removeSelectedVertices( edge.getFrom() );
155         removeSelectedVertices( edge.getTo() );
156     }

158     public void removeSelectedEdges(Collection<Edge> edges)      {
159         for ( Edge edge: edges )
160             removeSelectedEdges( edge );
161     }

163     @Override
164     protected void paintSetVertexStyle( Vertex vertex , Graphics2D g2d ) {
165         if ( isSelected( vertex ) ) {
166             g2d.setStroke( getSelectedEdgeStroke() );
167             g2d.setColor( getSelectedVertexColour() );
168         } else {
169             g2d.setStroke( getEdgeStroke() );
170             g2d.setColor( getVertexColour() );
171         }
172     }

174     @Override
175     protected void paintSetVertexFillStyle( Vertex vertex , Graphics2D g2d ) {
176         if ( isSelected( vertex ) ) {
177             g2d.setStroke( getSelectedEdgeStroke() );
178             g2d.setColor( getSelectedVertexFillColour() );
179         } else {
180             g2d.setStroke( getEdgeStroke() );
181             g2d.setColor( getVertexFillColour() );
182         }
183     }

185     @Override
186     protected void paintSetEdgeStyle( Edge edge , Graphics2D g2d ) {
187         if ( isSelected( edge ) ) {
188             g2d.setStroke( getSelectedEdgeStroke() );
189             g2d.setColor( getSelectedEdgeColour() );
190         } else if ( isConnected( edge ) ) {
191             g2d.setStroke( getConnectedEdgeStroke() );
192             g2d.setColor( getConnectedEdgeColour() );
193         } else {
194             g2d.setStroke( getEdgeStroke() );
195             g2d.setColor( getEdgeColour() );
196         }
197     }

199     @Override
200     public void paint( Graphics g ) {
201         super.paint( g );
202         if ( !clickedOnObject && clickLocation != null ) {
203             Graphics2D g2d = (Graphics2D) g;
204             g2d.setStroke( dragStroke );
205             g2d.setColor( dragColour );
206             int x, y, w, h;
207             if ( clickLocation.width >= 0 ) {
208                 x = clickLocation.x;
209                 w = clickLocation.width;
210             } else {
211                 x = clickLocation.x + clickLocation.width;
212                 w = -clickLocation.width;
213             }
214             if ( clickLocation.height >= 0 ) {
215                 y = clickLocation.y;
216                 h = clickLocation.height;
217             } else {
218                 y = clickLocation.y + clickLocation.height;

```



```

219         h = -clickLocation.height;
220     }
221     g2d.drawRect(x, y, w, h);
222 }
223 }

225 @Override
226 public void mousePressed( final MouseEvent event ) {
227     if ( bounds == null || !bounds.contains( event.getPoint() ) )
228         return;

230     clickLocation = new Rectangle( event.getPoint().x, event.getPoint().y, 0,
231     0 );
232     dragged = false;
233     clickedOnObject = false;
234     // objectStatus = DESELECTED;

235     final int modifiers = event.getModifiers();
236     final boolean ctrl = ( modifiers & InputEvent.CTRL_MASK ) != 0;
237     final boolean shift = ( modifiers & InputEvent.SHIFT_MASK ) != 0;
238     if ( !ctrl && !shift ) {
239         clearSelected();
240     }
241     final Vertex vertex = getVertexContaining( event.getPoint() );
242     if ( vertex != null ) {
243         if ( ctrl && isSelected( vertex ) ) {
244             removeSelectedVertices( vertex );
245         } else {
246             clickedOnObject = true;
247             addSelectedVertices( vertex );
248             // objectStatus = SELECTED;
249         }
250     } else {
251         final Edge edge = getEdgeContaining( event.getPoint() );
252         if ( edge != null ) {
253             if ( ctrl && isSelected( edge ) ) {
254                 removeSelectedEdges( edge );
255             } else {
256                 clickedOnObject = true;
257                 addSelectedEdges( edge );
258                 // objectStatus = SELECTED;
259             }
260         }
261     }
262     repaint();
263 }

265 @Override
266 public void mouseReleased( MouseEvent event ) {
267     if ( clickLocation == null )
268         return;
269     if ( clickedOnObject ) {
270         if ( dragged ) {
271             final double w = clickLocation.getWidth() / bounds.getWidth() *
272             100.0d;
273             final double h = clickLocation.getHeight() / bounds.getHeight() *
274             100.0d;
275             Point2D.Double offset = new Point2D.Double( w, h );
276             final HashSet<Vertex> vertices = new HashSet<Vertex>(
277             selectedVertices );
278             ConstantMovementMap map = new ConstantMovementMap( getGraphLayout(),
279             vertices, offset );
280             clickedOnObject = false;
281             clickLocation = null;
282             dragged = false;

```

```

279         setOffsetMovementMap( null );
280         getGraphLayout().processMovement(map);
281     }
282 } else {
283     HashSet<Vertex> vertices = getVerticesWithin( clickLocation );
284     HashSet<Edge> edges = getEdgesWithin( clickLocation );
285     if ( (event.getModifiers() & InputEvent.CTRL_MASK) != 0 ) {
286         if ( vertices != null )
287             for (Vertex vertex: vertices)
288                 if (isSelected( vertex ) ) {
289                     removeSelectedVertices(vertex);
290                     // System.err.println( "Removing: " + vertex.toString() );
291                 } else {
292                     addSelectedVertices(vertex);
293                     // System.err.println( "Adding: " + vertex.toString() );
294                 }
295             if ( edges != null )
296                 for (Edge edge: edges)
297                     if (isSelected( edge ) ) {
298                         removeSelectedEdges(edge);
299                         // System.err.println( "Removing: " + edge.toString() );
300                     } else {
301                         addSelectedEdges(edge);
302                         // System.err.println( "Adding: " + edge.toString() );
303                     }
304             } else {
305                 addSelectedVertices( vertices );
306                 addSelectedEdges( edges );
307             }
308         clickedOnObject = false;
309         clickLocation = null;
310         dragged = false;
311         repaint();
312     }
313 }

315 @Override
316 public void mouseDragged(MouseEvent event) {
317     if ( clickLocation == null )
318         return;
319     dragged = true;
320     Point p = event.getPoint();
321     clickLocation.width = p.x - clickLocation.x;
322     clickLocation.height = p.y - clickLocation.y;
323     if ( clickedOnObject ) {
324         final double w = clickLocation.getWidth() / bounds.getWidth() * 100.0d;
325         final double h = clickLocation.getHeight() / bounds.getHeight() *
326             100.0d;
327         Point2D.Double offset = new Point2D.Double( w, h );
328         ConstantMovementMap map = new ConstantMovementMap( getGraphLayout(),
329             selectedVertices, offset );
330         setOffsetMovementMap(map);
331     }
332     repaint();
333 }

```

D.5.4 mgtaylor.display.JIntegerTextField

```

1 package mgtaylor.display;
3 import java.awt.event.*;

```

```

4 import javax.swing.JTextField;
5 import javax.swing.text.Document;

7 public class JIntegerTextField extends JTextField {
8     private int maximum = 9999;

10     /**/
11     private static final long serialVersionUID = 6095737152358650784L;

13     public JIntegerTextField( final String text) {
14         super( Integer.toString( Integer.parseInt( text ) ) );
15     }

17     public JIntegerTextField( final int columns) {
18         super(columns);
19     }

21     public JIntegerTextField( final String text, final int columns) {
22         super( Integer.toString( Integer.parseInt( text ) ), columns );
23     }

25     public JIntegerTextField( final Document doc, final String text, final int
26         columns) {
27         super( doc, Integer.toString( Integer.parseInt( text ) ), columns );
28     }

29     public void processKeyEvent( final KeyEvent event ) {
30         final char c = event.getKeyChar();
31         if ( (Character.isDigit(c) && !event.isAltDown()) ) {
32             super.processKeyEvent( event );
33             if ( getValue() > maximum )
34                 setValue( maximum );
35         } else if ( ( c == 8 ) || ( c == 10 ) || ( c == 127 ) || ( c == 65535 ) ) {
36             /*
37              * 8:      Backspace
38              * 10:     Enter
39              * 127:    Delete
40              * 65535:  Direction Keys
41              */
42             super.processKeyEvent( event );
43         } else {
44             System.err.println( (int) c );
45             event.consume();
46         }
47     }

49     public int getValue() {
50         try {
51             return Integer.parseInt( getText() );
52         } catch (NumberFormatException e) {
53             return 0;
54         }
55     }

56     public void setValue( int value ) { setText( Integer.toString(
57         Math.min(maximum, value) ) ); }
58     public void setMaximum( int value ) { maximum = value; }

```

D.5.5 mgtaylor.display.PlanarityHandler

```

1 package mgtaylor.display;

```

```

3 import java.util.HashSet;

5 import mgtaylor.layout.VertexLayout;
6 import mgtaylor.planarity.event.*;

8 import mgtaylor.graph.*;

10 public class PlanarityHandler implements PlanarOrderHandler {
11     private final Graph graph;

13     private final HashSet<PlanarOrderListener> orderListeners = new
        HashSet<PlanarOrderListener>();

15     public PlanarityHandler(Graph graph) {
16         this.graph = graph;
17     }

19     public PlanarityHandler(VertexLayout layout) {
20         this(layout.getGraph());
21     }

23     public Graph getGraph() { return graph; }

25     public boolean isNonPlanarEdge(Edge edge) { return
        edge.hasPlanarityConflict(); }

27     @Override
28     public void addPlanarOrderListener(PlanarOrderListener listener) {
29         orderListeners.add(listener);
30     }

32     @Override
33     public void processPlanarOrderEvent(PlanarOrderEvent event) {
34         for (PlanarOrderListener listener : orderListeners)
35             listener.handlePlanarOrderChanged(event);
36     }

38     @Override
39     public void removePlanarOrderListener(PlanarOrderListener listener) {
40         orderListeners.remove(listener);
41     }
42 }

```

D.5.6 mgtaylor.display.PlanarityViewer

```

1 package mgtaylor.display;

3 import java.awt.*;
4 import java.awt.event.*;

6 import javax.swing.*;

8 import mgtaylor.display.data.*;
9 import mgtaylor.layout.*;

11 import mgtaylor.graph.*;
12 import mgtaylor.graph.graphTypes.OrderedFaceTrisectionGraph;
13 import mgtaylor.graph.graphTypes.PermutableFlippableGraph;
14 import mgtaylor.graph.graphTypes.PlanarMultiWheelGraph;
15 import mgtaylor.graph.graphTypes.PlanarWheelGraph;
16 import mgtaylor.graph.graphTypes.ReduceableBiconnectedGraphGenerator;
17 import mgtaylor.graph.graphTypes.RandomMaximalPlanarGraphGenerator;

```

```

18 import mgtaylor.graph.graphTypes.TriangularPrismMaximalPlanarGraph;
20 public class PlanarityViewer extends JFrame {
22     /** */
23     private static final long serialVersionUID = -6534711616529357001L;
25     public static final String GRAPH_HOPTAR = "nodes\n { A, B, C, D, E, F, G,
H, I, J, K, L, M, N, O, P }\nedges\n{\n (A,B), (B,C), (C,D), (D,E), (D,G),
(E,F), (F,G), (G,H), (H,F), (H,I), (I,J), (J,K), (L,A), (K,L), (L,M), (N,J),
(N,K), (M,N), (O,E), (O,M), (N,O), (B,P), (C,P), (K,P)\n}";
26     public static final String GRAPH_TEST1 = "nodes\n { A, B, C, D, E, F, G
}\nedges\n{\n (A,B), (A,C), (A,D),\n (A,E), (A,F), (A,G),\n (B,C), (C,D),
(C,F),\n (D,E), (D,G), (E,F),\n (E,G)\n}";
27     public static final String GRAPH_TEST2 = "nodes\n { A, B, C, D, E, F, G, H
}\nedges\n{\n (A,B), (A,C), (A,D),\n (A,E), (A,F), (A,G),\n (B,C), (C,D),
(C,F),\n (D,E), (D,G), (E,F),\n (E,G), (C,H), (H,A)\n}";
28     public static final String GRAPH_K5 = "nodes\n { A, B, C, D, E
}\nedges\n{\n (A,B), (A,C), (A,D), (A,E),\n (B,C), (B,D), (B,E),\n (C,D),
(C,E),\n (D,E)\n}";
29     public static final String GRAPH_K5_2 = "nodes\n { A, B, C, D, E
}\nedges\n{\n (A,B), (B,C), (C,D), (D,E),\n (E,A), (E,B), (E,C),\n (D,A),
(D,B),\n (C,A)\n}";
30     public static final String GRAPH_K3_3 = "nodes\n { A, B, C, D, E, F
}\nedges\n{\n (A,B), (A,D), (A,F),\n (C,B), (C,D), (C,F),\n (E,B), (E,D),
(E,F)\n}";
31     public static final String GRAPH_K3_3_2 = "nodes\n { A, B, C, D, E, F
}\nedges\n{\n (A,B), (B,C), (C,D),\n (D,E), (E,F), (F,A),\n (F,C), (E,B),
(D,A)\n}";
33     public static final String GRAPH_K3_3a = "nodes\n { O, A, B, C, D, E, F, G,
H, I }\nedges\n{\n (O,A), (A,B), (B,C), (C,D), (D,O),\n (D,E), (E,F),
(F,G), (G,A),\n (F,H), (H,B), (H,E),\n (F,I), (I,B), (I,C)}";
34     public static final String GRAPH_K3_3b = "nodes\n { O, A, B, C, D, E, F, G
}\nedges\n{\n (O,A), (A,B), (B,C), (C,D), (D,O),\n (D,E), (E,F), (F,G),
(G,A),\n (F,C), (E,B) }";
35     public static final String GRAPH_K3_3c = "nodes\n { O, A, B, C, D, E, F, G,
H }\nedges\n{\n (O,A), (A,B), (B,C), (C,O),\n (C,D), (D,E), (E,F), (F,G),
(G,A),\n (F,H), (H,B), (H,D),\n (E,B)}";
36     public static final String GRAPH_K3_3d = "nodes\n { O, A, B, C, D, E, F, G,
H, I }\nedges\n{\n (O,A), (A,B), (B,C), (C,D), (D,O),\n (D,E), (E,F),
(F,G), (G,H), (H,A),\n (G,I), (I,B), (I,E),\n (G,C)}";
37     public static final String GRAPH_K3_3e = "nodes\n { O, A, B, C, D, E, F, G,
H, I }\nedges\n{\n (O,A), (A,B), (B,C), (C,D), (D,O),\n (D,E), (E,F),
(F,G), (G,A),\n (F,H), (H,A), (H,C),\n (E,I), (I,A), (I,B)}";
38     public static final String GRAPH_K3_3f = "nodes\n { O, A, B, C, D, E, F, G,
H }\nedges\n{\n (O,A), (A,B), (B,C), (C,D), (D,O),\n (D,E), (E,F), (F,G),
(G,A),\n (F,H), (H,A), (H,C),\n (E,B)}";
39     public static final String GRAPH_K3_3g = "nodes\n { O, A, B, C, D, E, F, G,
H }\nedges\n{\n (O,A), (A,B), (B,C), (C,D), (D,O),\n (D,E), (E,F), (F,G),
(G,A),\n (E,H), (H,A), (H,C),\n (F,B)}";
41     private static final String DEFAULT_GRAPH_TEXT = GRAPH_HOPTAR; //GRAPH_K3_3g;
43     private final JTextArea data = new JTextArea( DEFAULT_GRAPH_TEXT );
44     private final JTabbedPane panes = new JTabbedPane();
45     private final JPanel generatorPanel = createGeneratorPanel();
46     private final JPanel inputPanel = createInputPanel();
47     private final DisplayPanel viewPanel = new DisplayPanel();
48     private final DisplayPanel dataPanel = new DisplayPanel();
49     private final DisplayPanel segmentPanel = new DisplayPanel();
50     private final DisplayPanel permutationPanel = new DisplayPanel();
51     private final DisplayPanel blockPanel = new DisplayPanel();
53     private Graph graph = null;

```

```

55 public PlanarityViewer() {
56     super();
57     setLayout(new BorderLayout());

59     try {
60         UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows.WindowsLookAndFeel");
61     } catch (Exception exception) {}

63     panes.addTab("Generate", generatorPanel);
64     panes.addTab("Input", inputPanel);
65     panes.addTab("Vertices", dataPanel);
66     panes.addTab("Segments", segmentPanel);
67     panes.addTab("Permutations", permutationPanel);
68     panes.addTab("Blocks", blockPanel);
69     JSplitPane split = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT, viewPanel,
70     panes);
71     split.setResizeWeight(0.5);
72     getContentPane().add(split, BorderLayout.CENTER);

73     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
74     pack();
75     setVisible(true);

77     panes.setSelectedComponent(inputPanel);
78 }

80 public Graph getGraph() {
81     return graph;
82 }

84 public void setGraph( final Graph graph ) {
85     setGraph( graph, 0 );
86 }

88 public void setGraph( final Graph graph, final int root ) {
89     this.graph = graph;
90     if ( graph != null ) {
91         final VertexLayout layout;
92         if ( graph instanceof PlanarWheelGraph || graph instanceof
93             PlanarMultiWheelGraph )
94             layout = new PlanarWheelLayout( graph );
95         else if ( graph instanceof TriangularPrismMaximalPlanarGraph )
96             layout = new TriangularPrismMaximalPlanarLayout( graph );
97         else if ( graph instanceof OrderedFaceTrisectionGraph )
98             layout = new OrderedVertexLayout( graph );
99         else if ( graph instanceof PermutableFlippableGraph )
100             layout = new PermutableFlippableLayout( graph );
101         else
102             layout = new RandomVertexLayout( graph );
103         PlanarityHandler planarity = new PlanarityHandler( layout );
104         final java.util.ArrayList<Vertex> vertices = graph.getVertices();
105         final int size = vertices.size();
106         graph.timePlanarityTest( vertices.get( Math.max( Math.min( root, size -
107             1), 0 ) ) );
108         System.out.println( graph.getSegments().getHead().toIndentedString( "",
109             " " ) );

110         viewPanel.setPanel( new AngleHighlightView( layout, planarity ) );
111         dataPanel.setPanel( new VertexDataView( layout, planarity ) );
112         segmentPanel.setPanel( new SegmentDataView( layout, planarity ) );
113         permutationPanel.setPanel( new PermutationDataView( layout, planarity ) );
114         blockPanel.setPanel( new BlockDataView( layout, planarity ) );
115         panes.setSelectedComponent( permutationPanel );

```

```

114         if (viewPanel.isVisible())
115             viewPanel.repaint();
116     }
117 }

119 public JPanel createGeneratorPanel() {
120     final JPanel panel = new JPanel();
121     panel.setLayout( new GridBagLayout() );

123     final JLabel typeLabel      = new JLabel( "Type" );
124     final JLabel verticesLabel  = new JLabel( "Vertices" );
125     final JLabel edgesLabel     = new JLabel( "Edges" );
126     final JLabel createLabel    = new JLabel( "Create ..." );

128     final JLabel maxPlanarLabel = new JLabel( "Maximum Planar:"
129 );
129     final JIntegerTextField maxPlanarVertices = new JIntegerTextField( "25", 4
130 );
130     final JButton createMaxPlanar = new JButton( "Create Max Planar
131 Graph ..." );
131     createMaxPlanar.addActionListener( new ActionListener() {
132         @Override
133         public void actionPerformed( ActionEvent arg0 ) {
134             graph = new RandomMaximalPlanarGraphGenerator(
135                 maxPlanarVertices.getValue() ).generateGraph(
136                 maxPlanarVertices.getValue() );
135             graph.setIdentifier( "Random Graph" );
136             setGraph( graph );
137             data.setText( graph.toString() );
138         }
139     });

141     final JLabel orderedPlanarLabel = new JLabel( "Ordered
142 Maximum Planar:" );
142     final JIntegerTextField orderedPlanarVertices = new JIntegerTextField(
143 "25", 4 );
143     final JButton createOrderedPlanar = new JButton( "Create
144 Ordered Planar Graph ..." );
144     createOrderedPlanar.addActionListener( new ActionListener() {
145         @Override
146         public void actionPerformed( ActionEvent arg0 ) {
147             graph = new OrderedFaceTrisectionGraph(
148                 orderedPlanarVertices.getValue() );
148             graph.setIdentifier( "Ordered Maximal Planar Graph" );
149             setGraph( graph );
150             data.setText( graph.toString() );
151         }
152     });

154     final JLabel triPrismLabel = new JLabel( "Triangular Prism:" );
155     final JIntegerTextField triPrismVertices = new JIntegerTextField( "25", 4
156 );
156     final JButton createTriPrismPlanar = new JButton( "Create Tri.
157 Prism Max Planar Graph ..." );
157     createTriPrismPlanar.addActionListener( new ActionListener() {
158         @Override
159         public void actionPerformed( ActionEvent arg0 ) {
160             graph = new TriangularPrismMaximalPlanarGraph(
161                 triPrismVertices.getValue() );
161             graph.setIdentifier( "Random Triangular Prism Graph" );
162             setGraph( graph );
163             data.setText( graph.toString() );
164         }
165     });

```

```

167     final JLabel planarWheelLabel          = new JLabel( "Planar Wheel Graph:"
168     );
169     final JTextField planarWheelVertices    = new JTextField(
170     "25", 4 );
171     final JButton createPlanarWheel        = new JButton( "Create
172     Graph..." );
173     createPlanarWheel.addActionListener(new ActionListener() {
174     @Override
175     public void actionPerformed(ActionEvent arg0) {
176         graph = new PlanarWheelGraph( planarWheelVertices.getValue() );
177         graph.setIdentifier("Random Graph");
178         setGraph( graph );
179         data.setText( graph.toString() );
180     }
181     });

182     final JLabel planarMultiWheelLabel      = new JLabel( "Planar
183     Wheel Graph:" );
184     final JTextField planarMultiWheelVertices = new
185     JTextField( "25", 4 );
186     final JTextField planarMultiWheelWheels = new
187     JTextField( "4", 4 );
188     final JCheckBox planarMultiWheelOuterCycle = new JCheckBox();
189     final JButton createMultiPlanarWheel    = new JButton( "Create
190     Graph..." );
191     createMultiPlanarWheel.addActionListener(new ActionListener() {
192     @Override
193     public void actionPerformed(ActionEvent arg0) {
194         graph = new PlanarMultiWheelGraph(
195             planarMultiWheelVertices.getValue(),
196             planarMultiWheelWheels.getValue(),
197             planarMultiWheelOuterCycle.isSelected()
198         );
199         graph.setIdentifier("Random Graph");
200         setGraph( graph );
201         data.setText( graph.toString() );
202     }
203     });

204     final JLabel biconnectedLabel          = new JLabel( "Biconnected:" );
205     final JTextField biconnectedVertices    = new JTextField( "5",
206     4 );
207     final JTextField biconnectedEdges      = new JTextField( "9", 5 );
208     final JButton createBiconnected        = new JButton( "Create
209     Biconnected Graph..." );
210     createBiconnected.addActionListener(new ActionListener() {
211     @Override
212     public void actionPerformed(ActionEvent arg0) {
213         int v = biconnectedVertices.getValue();
214         int e = biconnectedEdges.getValue();
215         Graph g = new RandomMaximalPlanarGraphGenerator( v ).generateGraph(
216             v );
217         graph = new ReduceableBiconnectedGraphGenerator( g,
218             g.getVertices().get( 0 ).generateGraph( g.edgeSize() - e );
219         graph.setIdentifier("Random Graph");
220         setGraph( graph );
221         data.setText( graph.toString() );
222     }
223     });

224     /*
225     final JLabel snakeLabel                = new JLabel( "Snake:" );
226     final JTextField snakeVertices          = new JTextField( "5", 4 );
227     final JTextField snakeRoot              = new JTextField( "5", 4 );
228     final JButton createSnake               = new JButton( "Create Snake
229     Graph..." );

```



```

220         createSnake.addActionListener(new ActionListener() {
221             @Override
222             public void actionPerformed(ActionEvent arg0) {
223                 final int v = snakeVertices.getValue();
224                 graph = new mgtaylor.graph.graphTypes.SnakeGraph(v);
225                 graph.setIdentifier("Snake Graph");
226                 setGraph( graph, snakeRoot.getValue() - 1 );
227                 data.setText(graph.toString());
228             }
229         });
230     */

232     final JLabel pfLabel = new JLabel( "Permutable Flippable:" );
233     final JTextField pfVertices = new JTextField("5", 4);
234     final JTextField pfRoot = new JTextField("1", 4);
235     final JButton createPF = new JButton( "Create Perm. Flip.
Graph..." );
236     createPF.addActionListener(new ActionListener() {
237         @Override
238         public void actionPerformed(ActionEvent arg0) {
239             final int v = pfVertices.getValue();
240             graph = new mgtaylor.graph.graphTypes.PermutableFlippableGraph(v);
241             graph.setIdentifier("Permutable Flippable Graph");
242             setGraph( graph, pfRoot.getValue() - 1 );
243             data.setText(graph.toString());
244         }
245     });

247     int y = 0;
248     panel.add(typeLabel, getGBC(0, y, 1, 1, 0.0, 0.0));
249     panel.add(verticesLabel, getGBC(1, y, 1, 1, 1.0,
0.0));
250     panel.add(edgesLabel, getGBC(2, y, 2, 1, 1.0,
0.0));
251     panel.add(createLabel, getGBC(4, y++, 1, 1, 0.0,
0.0));
252     panel.add(new JSeparator( JSeparator.HORIZONTAL), getGBC(0, y++, 5, 1,
0.0, 0.0));
253     panel.add(maxPlanarLabel, getGBC(0, y, 1, 1, 0.0,
0.0));
254     panel.add(maxPlanarVertices, getGBC(1, y, 1, 1, 1.0,
0.0));
255     panel.add(createMaxPlanar, getGBC(4, y++, 1, 1, 0.0,
0.0));
256     panel.add(orderedPlanarLabel, getGBC(0, y, 1, 1, 0.0,
0.0));
257     panel.add(orderedPlanarVertices, getGBC(1, y, 1, 1, 1.0,
0.0));
258     panel.add(createOrderedPlanar, getGBC(4, y++, 1, 1, 0.0,
0.0));
259     panel.add(triPrismLabel, getGBC(0, y, 1, 1, 0.0,
0.0));
260     panel.add(triPrismVertices, getGBC(1, y, 1, 1, 1.0,
0.0));
261     panel.add(createTriPrismPlanar, getGBC(4, y++, 1, 1, 0.0,
0.0));
262     panel.add(planarWheelLabel, getGBC(0, y, 1, 1, 0.0,
0.0));
263     panel.add(planarWheelVertices, getGBC(1, y, 1, 1, 1.0,
0.0));
264     panel.add(createPlanarWheel, getGBC(4, y++, 1, 1, 0.0,
0.0));
265     panel.add(planarMultiWheelLabel, getGBC(0, y, 1, 1, 0.0,
0.0));

```

```

266     panel.add(planarMultiWheelVertices,      getGBC(1, y,    1, 1, 1.0,
267             0.0));
268     panel.add(planarMultiWheelWheels,        getGBC(2, y,    1, 1, 1.0,
269             0.0));
270     panel.add(planarMultiWheelOuterCycle,    getGBC(3, y,    1, 1, 1.0,
271             0.0));
272     panel.add(createMultiPlanarWheel,        getGBC(4, y++, 1, 1, 0.0,
273             0.0));
274     panel.add(biconnectedLabel,              getGBC(0, y,    1, 1, 0.0,
275             0.0));
276     panel.add(biconnectedVertices,           getGBC(1, y,    1, 1, 1.0,
277             0.0));
278     panel.add(biconnectedEdges,              getGBC(2, y,    2, 1, 1.0,
279             0.0));
280     panel.add(createBiconnected,             getGBC(4, y++, 1, 1, 0.0,
281             0.0));
282     panel.add(pfLabel,                       getGBC(0, y,    1, 1, 0.0,
283             0.0));
284     panel.add(pfVertices,                    getGBC(1, y,    1, 1, 1.0,
285             0.0));
286     panel.add(pfRoot,                        getGBC(2, y,    2, 1, 1.0, 0.0));
287     panel.add(createPF,                      getGBC(4, y++, 1, 1, 0.0,
288             0.0));
289     // panel.add(snakeLabel,                  getGBC(0, y,    1, 1, 0.0,
290             0.0));
291     // panel.add(snakeVertices,               getGBC(1, y,    1, 1, 1.0,
292             0.0));
293     // panel.add(snakeRoot,                   getGBC(2, y,    1, 1, 1.0, 0.0));
294     // panel.add(createSnake,                 getGBC(4, y++, 1, 1, 0.0,
295             0.0));
296     return panel;
297 }

298
299 public JPanel createInputPanel() {
300     final JPanel panel = new JPanel();
301     panel.setLayout(new GridBagLayout());
302
303     final JButton load = new JButton("Load");
304     load.addActionListener(new ActionListener() {
305         @Override
306         public void actionPerformed(ActionEvent arg0) {
307             setGraph( new Graph(data.getText()) );
308         }
309     });
310
311     final JButton speedTest = new JButton("Speed Test");
312     speedTest.addActionListener(new ActionListener() {
313         @Override
314         public void actionPerformed(ActionEvent arg0) {
315             int[] vertexNumbers = {5000, 1000}; // 500, 200, 100, 75, 50, 40,
316             30, 20, 10
317             for (int j: vertexNumbers) {
318                 System.out.println(j);
319                 System.out.println();
320                 for (int i = 0; i < 1000; i++) {
321                     graph = new
322                         RandomMaximalPlanarGraphGenerator(j).generateGraph(j);
323                     graph.setIdentifier(Integer.toString(i));
324                     graph.timePlanarityTest();
325                     graph = null;
326                     if (i % (5000/j) == 0) System.gc();
327                 }
328             }
329         }
330     });
331 }

```

```

315     panel.add(new JScrollPane(data), getGBC(0, 0, 2, 1, 1.0, 1.0));
316     panel.add(load, getGBC(0, 1, 1, 1, 1.0, 0.0));
317     panel.add(speedTest, getGBC(1, 1, 1, 1, 1.0, 0.0));
318     return panel;
319 }

321 public static GridBagConstraints getGBC(int x, int y, int w, int h, double
wx, double wy) {
322     GridBagConstraints gbc = new GridBagConstraints();
323     gbc.gridx = x;
324     gbc.gridy = y;
325     gbc.ipadx = 0;
326     gbc.ipady = 0;
327     gbc.insets = new Insets(2, 2, 2, 2);
328     gbc.weightx = wx;
329     gbc.weighty = wy;
330     gbc.gridwidth = w;
331     gbc.gridheight = h;
332     gbc.fill = GridBagConstraints.BOTH;
333     gbc.anchor = GridBagConstraints.CENTER;
334     return gbc;
335 }

337 /**
338  * @param args
339  */
340 public static void main(String[] args) {
341     new PlanarityViewer();
342 }

344 public static class DisplayPanel extends JPanel {
345     /**/
346     private static final long serialVersionUID = -1722530348771628881L;
347     private JPanel panel = null;

349     public DisplayPanel() {
350         super();
351         setLayout(new BorderLayout());
352         setMinimumSize(new Dimension(400, 400));
353         setPreferredSize(new Dimension(400, 400));
354     }

356     public void setPanel(JPanel panel) {
357         if (this.panel != null) {
358             remove(this.panel);
359             if (this.panel instanceof KeyListener)
360                 removeKeyListener((KeyListener) this.panel);
361         }
362         this.panel = panel;
363         if (panel != null) {
364             add(panel, BorderLayout.CENTER);
365             if (this.panel instanceof KeyListener)
366                 addKeyListener((KeyListener) panel);
367         }
368     }
369 }
370 }

```

D.5.7 mgtaylor.display.View

```

1 package mgtaylor.display;

```

```

3 import java.awt.*;
4 import java.awt.event.*;
5 import java.awt.font.*;
6 import java.awt.geom.*;
7 import java.util.*;
8 import javax.swing.*;

10 import mgtaylor.layout.*;
11 import mgtaylor.layout.event.*;

13 import mgtaylor.graph.*;

15 /**
16  * View is a layer of abstraction between a component displaying the
17  * visualisation of the graph and a graph data structure.
18  *
19  * The view maintains a set of co-ordinates for each vertex (With
20  * values 0.0d to 100.0d for the x and y co-ordinates), which are
21  * independent of the co-ordinate system used by the displaying
22  * component. When a component is displayed the vertex co-ordinates
23  * are converted to screen locations, meaning that operations can be
24  * performed on the view irrespective of the changes to the actual
25  * output area.
26  *
27  * Output region is a square canvas of maximum possible size for the
28  * displaying component. This prevents distortion due to scaling on a
29  * single axis.
30  *
31  * Pressing the keys Ctrl-z will undo vertex movements.
32  * Pressing the keys Ctrl-y will redo vertex movements.
33  *
34  * @author Martyn G Taylor
35  */
36
37 public abstract class View
38     extends JPanel
39     implements ComponentListener, GraphLayoutListener,
40         MouseListener, MouseMotionListener, KeyListener {
41     /***/
42     private static final long serialVersionUID = -6879984664846100587L;

44     private static final Dimension MINIMUM_SIZE      = new Dimension( 100, 100 );
45     private static final Dimension PREFERRED_SIZE    = new Dimension( 100, 100 );
46     private static final Dimension MAXIMUM_SIZE      = null;

48     private final VertexLayout layout;
49     private final View view;

51     private final HashMap<Vertex, Shape> vertexShapeMap      = new
HashMap<Vertex, Shape>();
52     private final HashMap<Edge, Shape> edgeShapeMap          = new
HashMap<Edge, Shape>();
53     private final HashMap<Vertex, Shape> offsetVertexShapeMap = new
HashMap<Vertex, Shape>();
54     private final HashMap<Edge, Shape> offsetEdgeShapeMap     = new
HashMap<Edge, Shape>();
55     private MovementMap vertexOffsetMap                      = null;

57     protected Double vertexRadius = 3.0d;
58     protected Double edgeRadius = 2.0d;

60     private Color backgroundColour      = Color.white;
61     private BasicStroke edgeStroke      = new BasicStroke(1.0f);
62     private BasicStroke vertexStroke    = new BasicStroke(1.0f);
63     private boolean showVertexLabel     = true;

```

```

64  private Color edgeColour          = Color.black;
65  private Color vertexColour        = Color.black;
66  private Color vertexFillColour    = getBackgroundColour();
67  private Color textColour          = Color.black;
68  private Font labelFont            = new Font( "Arial", Font.BOLD, 10 );

70  private BasicStroke offsetEdgeStroke    = new BasicStroke(1.0f);
71  private BasicStroke offsetVertexStroke = new BasicStroke(1.0f);
72  private Color offsetEdgeColour          = Color.blue;
73  private Color offsetVertexColour        = Color.blue;
74  private Color offsetVertexFillColour    = getBackgroundColour();
75  private Color offsetTextColour          = Color.blue;
76  private Font offsetLabelFont            = getFont();

78  protected Rectangle bounds;
79  protected BasicStroke outlineStroke = new BasicStroke(1.0f);
80  protected Color outlineColour = Color.BLACK;

82  public View(VertexLayout layout) {
83      super();
84      this.layout = layout;
85      view = null;
86      bounds = getBounds();
87      setSelfListeners();
88      if ( MINIMUM_SIZE != null )      setMinimumSize(MINIMUM_SIZE);
89      if ( PREFERRED_SIZE != null )   setPreferredSize(PREFERRED_SIZE);
90      if ( MAXIMUM_SIZE != null )     setMaximumSize(MAXIMUM_SIZE);
91  }

93  public View(View view) {
94      super();
95      layout = view.getGraphLayout();
96      this.view = view;
97      bounds = getBounds();
98      setSelfListeners();
99  }

101 private void setSelfListeners() {
102     addMouseListener( this );
103     addMouseMotionListener( this );
104     addKeyListener( this );
105     addComponentListener( this );
106     layout.addGraphLayoutListener( this );
107 }

109 public VertexLayout getGraphLayout()      { return layout; }
110 public View getPreviousView()             { return view; }

112 protected abstract void generateInitialData();

114 protected Shape getVertexShape( final Vertex vertex ) {
115     return vertexShapeMap.get(vertex);
116 }

118 protected void setVertexShape( final Vertex vertex, final Shape shape ) {
119     if ( vertexShapeMap.containsKey(vertex) )
120         vertexShapeMap.remove(vertex);
121     vertexShapeMap.put(vertex, shape);
122 }

124 protected Shape getEdgeShape( final Edge edge ) {
125     return edgeShapeMap.get(edge);
126 }

128 protected void setEdgeShape( final Edge edge, final Shape shape ) {

```

```

129         if (edgeShapeMap.containsKey(edge))
130             edgeShapeMap.remove(edge);
131         if ( shape != null )
132             edgeShapeMap.put(edge, shape);
133     }

135     protected Shape generateVertexShape(final Vertex vertex) {
136         if (vertex == null)
137             return null;
138         final Point2D.Double p = invertComponentLocation(
139             getGraphLayout().getVertexLocation(vertex) );
140         return new Ellipse2D.Double(
141             p.x - vertexRadius,
142             p.y - vertexRadius,
143             2 * vertexRadius, 2 * vertexRadius
144         );
145     }

146     protected Shape generateEdgeShape(final Point2D.Double from, final
147     Point2D.Double to ) {
148         if (from == null || to == null)
149             return null;
150         Point2D.Double v1 = invertComponentLocation( from );
151         Point2D.Double v2 = invertComponentLocation( to );
152         return new Line2D.Double( v1.x, v1.y, v2.x, v2.y );
153     }

154     protected Shape generateEdgeShape(final Edge edge) {
155         if (edge == null)
156             return null;
157         Point2D.Double v1 = getGraphLayout().getVertexLocation(edge.getFrom());
158         Point2D.Double v2 = getGraphLayout().getVertexLocation(edge.getTo());
159         return generateEdgeShape( v1, v2 );
160     }

162     protected Shape getOffsetVertexShape( final Vertex vertex ) {
163         return offsetVertexShapeMap.get(vertex);
164     }

166     protected void setOffsetVertexShape( final Vertex vertex, final Shape shape )
167     {
168         if (offsetVertexShapeMap.containsKey(vertex))
169             offsetVertexShapeMap.remove(vertex);
170         if ( shape != null )
171             offsetVertexShapeMap.put(vertex, shape);
172     }

173     protected Shape getOffsetEdgeShape( final Edge edge ) {
174         return offsetEdgeShapeMap.get(edge);
175     }

177     protected void setOffsetEdgeShape( final Edge edge, final Shape shape ) {
178         if (offsetEdgeShapeMap.containsKey(edge))
179             offsetEdgeShapeMap.remove(edge);
180         if ( shape != null )
181             offsetEdgeShapeMap.put(edge, shape);
182     }

184     protected Shape generateOffsetVertexShape( final Vertex vertex ) {
185         if ( vertex == null || vertexOffsetMap == null ||
186             !vertexOffsetMap.containsKey(vertex))
187             return null;
188         final Point2D.Double offset = vertexOffsetMap.getVertexOffset(vertex);
189         if ( offset == null ) return null;

```

```

189         final Point2D.Double p = invertComponentLocation(
190             getGraphLayout().getVertexLocation(vertex, offset) );
191         return new Ellipse2D.Double(
192             p.x - vertexRadius,
193             p.y - vertexRadius,
194             2 * vertexRadius, 2 * vertexRadius
195         );
196     }
197
198     protected Shape generateOffsetEdgeShape(final Edge edge) {
199         if (edge == null || vertexOffsetMap == null ||
200             (!vertexOffsetMap.containsKey(edge.getFrom()) &&
201              !vertexOffsetMap.containsKey(edge.getTo())))
202             return null;
203         Point2D.Double v1;
204         if (vertexOffsetMap.containsKey(edge.getFrom()))
205             v1 = getGraphLayout().getVertexLocation(edge.getFrom(),
206                 vertexOffsetMap.getVertexOffset(edge.getFrom()));
207         else
208             v1 = getGraphLayout().getVertexLocation(edge.getFrom());
209         Point2D.Double v2;
210         if (vertexOffsetMap.containsKey(edge.getTo()))
211             v2 = getGraphLayout().getVertexLocation(edge.getTo(),
212                 vertexOffsetMap.getVertexOffset(edge.getTo()));
213         else
214             v2 = getGraphLayout().getVertexLocation(edge.getTo());
215         return generateEdgeShape(v1, v2);
216     }
217
218     public void setOffsetMovementMap( final MovementMap map ) {
219         offsetEdgeShapeMap.clear();
220         offsetVertexShapeMap.clear();
221         if (map == null) {
222             vertexOffsetMap = null;
223             return;
224         }
225         if (map.getLayout() != getGraphLayout())
226             return;
227         this.vertexOffsetMap = map;
228         final Graph graph = getGraphLayout().getGraph();
229         HashSet<Edge> edges = new HashSet<Edge>();
230         for ( final Vertex vertex: map.getVertices() )
231             if ( graph.contains(vertex) ) {
232                 setOffsetVertexShape( vertex, generateOffsetVertexShape( vertex ) );
233                 for ( Edge edge: vertex.getConnectedEdges() )
234                     if ( !edges.contains(edge) ) {
235                         setOffsetEdgeShape( edge, generateOffsetEdgeShape( edge ) );
236                         edges.add(edge);
237                     }
238             }
239     }
240
241     public Color getBackgroundColour() { return backgroundColour; }
242     public boolean isShowingVertexLabel() { return showVertexLabel; }
243     public BasicStroke getEdgeStroke() { return edgeStroke; }
244     public BasicStroke getVertexStroke() { return vertexStroke; }
245     public Color getEdgeColour() { return edgeColour; }
246     public Color getVertexColour() { return vertexColour; }
247     public Color getVertexFillColour() { return vertexFillColour; }
248     public Color getTextColour() { return textColour; }
249     public Font getFont() { return labelFont; }
250     public BasicStroke getOutlineStroke() { return outlineStroke; }
251     public Color getOutlineColour() { return outlineColour; }
252     public Rectangle getBounds() { return bounds; }

```

```

250 public BasicStroke getOffsetEdgeStroke() { return offsetEdgeStroke; }
251 public BasicStroke getOffsetVertexStroke() { return offsetVertexStroke; }
252 public Color getOffsetEdgeColour() { return offsetEdgeColour; }
253 public Color getOffsetVertexColour() { return offsetVertexColour; }
254 public Color getOffsetVertexFillColour() { return offsetVertexFillColour; }
255 public Color getOffsetTextColour() { return offsetTextColour; }
256 public Font getOffsetFont() { return offsetLabelFont; }

258 public void setShowVertexLabel(boolean show) { showVertexLabel =
show; }
259 public void setEdgeStroke(BasicStroke stroke) { edgeStroke =
stroke; }
260 public void setVertexStroke(BasicStroke stroke) { vertexStroke =
stroke; }
261 public void setEdgeColour(Color colour) { edgeColour =
colour; }
262 public void setVertexColour(Color colour) { vertexColour =
colour; }
263 public void setVertexFillColour(Color colour) { vertexFillColour =
colour; }
264 public void setTextColour(Color colour) { textColour =
colour; }
265 public void setOutlineStroke(BasicStroke stroke) { outlineStroke =
stroke; }
266 public void setOutlineColour(Color colour) { outlineColour =
colour; }
267 public void setFont(String name, int style, int size) { setFont( new Font(
name, style, size)); }
268 public void setFont(Font font) { labelFont = font; }

270 public void setOffsetEdgeStroke(BasicStroke stroke) {
offsetEdgeStroke = stroke; }
271 public void setOffsetVertexStroke(BasicStroke stroke) { offsetVertexStroke
= stroke; }
272 public void setOffsetEdgeColour(Color colour) { offsetEdgeColour =
colour; }
273 public void setOffsetVertexColour(Color colour) { offsetVertexColour
= colour; }
274 public void setOffsetVertexFillColour(Color colour) {
offsetVertexFillColour = colour; }
275 public void setOffsetTextColour(Color colour) { offsetTextColour =
colour; }
276 public void setOffsetFont(Font font) { offsetLabelFont =
font; }
277 public void setOffsetFont(String name, int style, int size) {
setOffsetFont(new Font( name, style, size )); }

279 protected Point2D.Double translateComponentLocation( Point p ) {
280     if ( p.x < bounds.getMinX() || p.y < bounds.getMinY() ||
281         p.x > bounds.getMaxX() || p.y > bounds.getMaxY() )
282         return null;
283     return translateComponentLocationIgnoreBounds( p );
284 }

286 protected Point2D.Double translateComponentLocationIgnoreBounds( Point p ) {
287     double x = ( p.x - bounds.x ) * 100.0d / (double) bounds.width;
288     double y = ( p.y - bounds.y ) * 100.0d / (double) bounds.height;
289     return new Point2D.Double( x, y );
290 }

292 protected Point2D.Double invertComponentLocation( Point2D.Double p ) {
293     return new Point2D.Double( bounds.width * p.x / 100, bounds.height * p.y /
100 );
294 }

```



```

296     protected boolean contains(Point2D.Double p, Vertex vertex) {
297         if (vertex != null) {
298             Point2D.Double v = getGraphLayout().getVertexLocation(vertex);
299             double dx = v.x - p.x;
300             double dy = v.y - p.y;
301             return dx * dx + dy * dy <= vertexRadius * vertexRadius;
302         }
303         return false;
304     }

306     protected boolean contains(Point2D.Double p, Edge edge) {
307         if (edge != null) {
308             Point2D.Double f = getGraphLayout().getVertexLocation(edge.getFrom());
309             Point2D.Double t = getGraphLayout().getVertexLocation(edge.getTo());
310             if (f.x == t.x && f.y == t.y) {
311                 return (p.x - f.x) * (p.x - f.x) + (p.y - f.y) * (p.y - f.y) <=
312                     edgeRadius * edgeRadius );
313             } else {
314                 double tx = t.x - f.x;
315                 double ty = t.y - f.y;
316                 double px = p.x - f.x;
317                 double py = p.y - f.y;
318                 double d = Math.sqrt(tx * tx + ty * ty);
319                 double sin = ty / d;
320                 double cos = tx / d;
321                 double x = cos * px + sin * py;
322                 double y = cos * py - sin * px;
323                 if (Math.abs(y) > edgeRadius)
324                     return false;
325                 if (x < 0) {
326                     return (x * x + y * y <= edgeRadius * edgeRadius);
327                 } else if (x > d) {
328                     return ((x - d) * (x - d) + y * y <= edgeRadius * edgeRadius);
329                 } else {
330                     return true;
331                 }
332             }
333         }
334         return false;
335     }

336     protected Vertex getVertexContaining(Point p) {
337         Point2D.Double p2d = translateComponentLocation(p);
338         for (Vertex vertex: getGraphLayout().getGraph().getVertices())
339             if (contains(p2d, vertex))
340                 return vertex;
341         return null;
342     }

344     protected Edge getEdgeContaining(Point p) {
345         Point2D.Double p2d = translateComponentLocation(p);
346         for (Edge edge: getGraphLayout().getGraph().getEdges())
347             if (contains(p2d, edge))
348                 return edge;
349         return null;
350     }

352     protected boolean isBetween(final Vertex vertex, final Point2D.Double p1,
353     final Point2D.Double p2) {
354         Point2D.Double v = getGraphLayout().getVertexLocation(vertex);
355         return (p1.x <= v.x) && (v.x <= p2.x) && (p1.y <= v.y) && (v.y <
356         p2.y);

```

```

357     protected boolean isBetween(final Edge edge, final Point2D.Double p1, final
Point2D.Double p2) {
358         return isBetween( edge.getFrom(), p1, p2 ) && isBetween( edge.getTo(), p1,
p2 );
359     }

361     protected HashSet<Vertex> getVerticesWithin(final Rectangle r) {
362         if ( r == null )
363             return null;
364         final HashSet<Vertex> matchedVertices = new HashSet<Vertex>();
365         Point2D.Double p1 = translateComponentLocationIgnoreBounds( new Point(
(int) r.getMinX(), (int) r.getMinY() ) );
366         Point2D.Double p2 = translateComponentLocationIgnoreBounds( new Point(
(int) r.getMaxX(), (int) r.getMaxY() ) );
367         for (Vertex vertex: getGraphLayout().getGraph().getVertices())
368             if ( isBetween( vertex, p1, p2 ) )
369                 matchedVertices.add( vertex );
370         return matchedVertices;
371     }

373     protected HashSet<Edge> getEdgesWithin(final Rectangle r) {
374         if ( r == null )
375             return null;
376         final HashSet<Edge> matchedVertices = new HashSet<Edge>();
377         Point2D.Double p1 = translateComponentLocationIgnoreBounds( new Point(
(int) r.getMinX(), (int) r.getMinY() ) );
378         Point2D.Double p2 = translateComponentLocationIgnoreBounds( new Point(
(int) r.getMaxX(), (int) r.getMaxY() ) );
379         for (Edge edge: getGraphLayout().getGraph().getEdges())
380             if ( isBetween( edge, p1, p2 ) )
381                 matchedVertices.add( edge );
382         return matchedVertices;
383     }

386     protected void paintBorder( Graphics2D g2d ) {
387         int border = (int) Math.ceil( outlineStroke.getLineWidth() / 2 );
388         g2d.setStroke( outlineStroke );
389         g2d.setColor( outlineColour );
390         g2d.drawRect( -border, -border, bounds.width + border, bounds.height +
border );
391     }

393     protected void paintSetVertexStyle( Vertex vertex, Graphics2D g2d ) {
394         g2d.setStroke( getEdgeStroke() );
395         g2d.setColor( getVertexColour() );
396     }

398     protected void paintSetVertexFillStyle( Vertex vertex, Graphics2D g2d ) {
399         g2d.setStroke( getEdgeStroke() );
400         g2d.setColor( getVertexFillColour() );
401     }

403     protected void paintSetOffsetVertexStyle( Vertex vertex, Graphics2D g2d ) {
404         g2d.setStroke( getOffsetEdgeStroke() );
405         g2d.setColor( getOffsetVertexColour() );
406     }

408     protected void paintSetOffsetVertexFillStyle( Vertex vertex, Graphics2D g2d )
{
409         g2d.setStroke( getOffsetEdgeStroke() );
410         g2d.setColor( getOffsetVertexFillColour() );
411     }

413     protected void paintSetEdgeStyle( Edge edge, Graphics2D g2d ) {

```

```

414         g2d.setStroke(getEdgeStroke());
415         g2d.setColor(getEdgeColour());
416     }

418     protected void paintSetOffsetEdgeStyle( Edge edge, Graphics2D g2d ) {
419         g2d.setStroke(getOffsetEdgeStroke());
420         g2d.setColor(getOffsetEdgeColour());
421     }

423     protected void paintVertex( Vertex vertex, Graphics2D g2d ) {
424         Shape shape = getVertexShape( vertex );
425         paintSetVertexFillStyle( vertex, g2d );
426         g2d.fill(shape);
427         paintSetVertexStyle( vertex, g2d );
428         g2d.draw(shape);
429     }

431     protected void paintOffsetVertex( Vertex vertex, Graphics2D g2d ) {
432         Shape shape = getOffsetVertexShape( vertex );
433         paintSetOffsetVertexFillStyle( vertex, g2d );
434         g2d.fill(shape);
435         paintSetOffsetVertexStyle( vertex, g2d );
436         g2d.draw(shape);
437     }

439     protected void paintVertexLabel( Vertex vertex, Graphics2D g2d ) {
440         Shape shape = getVertexShape( vertex );
441         if (!vertex.toString().equals("") && isShowingVertexLabel()) {
442             g2d.setColor(getTextColour());
443             Font font = getFont();
444             Rectangle bounds = shape.getBounds();
445             int x = bounds.x + bounds.width + 2;
446             int y = bounds.y + bounds.height + 2;

448             FontRenderContext frc = g2d.getFontRenderContext();
449             TextLayout labelLayout = new TextLayout(vertex.toString(), font, frc);
450             labelLayout.draw(g2d, x, y);
451         }
452     }

454     protected void paintEdge( Edge edge, Graphics2D g2d ) {
455         paintSetEdgeStyle( edge, g2d );
456         g2d.draw(getEdgeShape(edge));
457     }

459     protected void paintOffsetEdge( Edge edge, Graphics2D g2d ) {
460         paintSetOffsetEdgeStyle( edge, g2d );
461         g2d.draw(getOffsetEdgeShape(edge));
462     }

464     protected void beforePaint( final Graphics2D g2d ) {
465         g2d.setColor( getBackgroundColour() );
466         g2d.fillRect( 0, 0, getWidth(), getHeight());
467         g2d.translate( bounds.x, bounds.y);
468         if (outlineStroke != null)
469             paintBorder( g2d );
470         g2d.setClip(0, 0, bounds.width, bounds.height);
471     }

473     protected void afterPaint( final Graphics2D g2d ) {
474         g2d.translate( -bounds.x, -bounds.y);
475     }
476     protected void paint( final Graphics g, final boolean paintBackground ) {
477         if ( bounds == null )
478             return;

```

```

479     Graphics2D g2d = (Graphics2D) g;
480     if ( paintBackground ) {
481         beforePaint( g2d );
482     }
483     ArrayList<Edge> edges = getGraphLayout().getGraph().getEdges();
484     ArrayList<Vertex> vertices = getGraphLayout().getGraph().getVertices();
485     for (Edge edge: edges) {
486         paintEdge( edge, g2d );
487         if ( vertexOffsetMap != null &&
488             (vertexOffsetMap.contains(edge.getTo()) ||
489              vertexOffsetMap.contains(edge.getFrom()))) {
489             paintOffsetEdge( edge, g2d );
490         }
491     }
492     for (Vertex vertex: vertices) {
493         paintVertex( vertex, g2d );
494         paintVertexLabel( vertex, g2d );
495         if ( vertexOffsetMap != null && vertexOffsetMap.contains(vertex))
496             paintOffsetVertex( vertex, g2d );
497     }
498     if ( paintBackground ) {
499         afterPaint( g2d );
500     }
501 }
502 public void paint(Graphics g) {
503     paint( g, true );
504 }
505
506 protected void regenerateView( MovementMap map ) {
507     for (Vertex vertex: getGraphLayout().getGraph().getVertices())
508         if ( map == null || map.contains(vertex) ) {
509             setVertexShape( vertex, generateVertexShape( vertex ) );
510             setOffsetVertexShape( vertex, generateOffsetVertexShape( vertex ) );
511         }
512     for ( Edge edge: getGraphLayout().getGraph().getEdges() )
513         if ( map == null || map.contains( edge.getFrom() ) || map.contains(
514             edge.getTo() ) ) {
515             setEdgeShape( edge, generateEdgeShape( edge ) );
516             setOffsetEdgeShape( edge, generateOffsetEdgeShape( edge ) );
517         }
518 }
519
520 public void regenerateBounds() {
521     Dimension d = getSize();
522     int border = (int) Math.ceil( outlineStroke.getLineWidth() );
523     bounds = new Rectangle( border, border, d.width - 2 * border, d.height - 2
524         * border );
525     if (d.width > d.height)
526         bounds = new Rectangle(
527             (d.width - d.height) / 2 + border,
528             border,
529             d.height - 2 * border,
530             d.height - 2 * border
531         );
532     else
533         bounds = new Rectangle(
534             border,
535             (d.height - d.width) / 2 + border,
536             d.width - 2 * border,
537             d.width - 2 * border
538         );
539     // System.err.println( "View.regenerateBounds(): (" + d.width + ", " +
540     // d.height + " ), (" + bounds.x + ", " + bounds.y + ", " + bounds.width + ", " +
541     // bounds.height + "]" );
542     regenerateView( null );

```

```

539     repaint();
540 }

542 @Override
543 public void handleGraphLayoutChanged(GraphLayoutEvent event) {
544     regenerateView( event.getMovementMap() );
545     repaint();
546 }

548 @Override
549 public void componentResized(ComponentEvent event) {
550     regenerateBounds();
551 }

553 @Override
554 public void keyTyped(KeyEvent event) {
555     switch ( event.getKeyChar() ) {
556         case 26: getGraphLayout().undo(); break; // Ctrl-z
557         case 25: getGraphLayout().redo(); // Ctrl-y
558         default:
559             // System.err.println( (int) event.getKeyChar() );
560     }
561 }

563 @Override public void componentHidden(ComponentEvent event) {}
564 @Override public void componentMoved(ComponentEvent event) {}
565 @Override public void componentShown(ComponentEvent event) {}
566 @Override public void mouseClicked(MouseEvent arg0) {}
567 @Override public void mouseEntered(MouseEvent arg0) {}
568 @Override public void mouseExited(MouseEvent arg0) {}
569 @Override public void mousePressed(MouseEvent arg0) {}
570 @Override public void mouseReleased(MouseEvent arg0) {}
571 @Override public void mouseDragged(MouseEvent arg0) {}
572 @Override public void mouseMoved(MouseEvent arg0) {}
573 @Override public void keyPressed(KeyEvent arg0) {}
574 @Override public void keyReleased(KeyEvent arg0) {}
575 }

```

D.5.8 mgtaylor.display.data.BlockDataView

```

1 package mgtaylor.display.data;

3 import java.awt.*;

5 import javax.swing.*;
6 import javax.swing.table.*;

8 import mgtaylor.display.PlanarityHandler;
9 import mgtaylor.display.data.event.*;

11 import mgtaylor.datastructures.UnmodifiableLinkedList;
12 import mgtaylor.graph.Block;
13 import mgtaylor.graph.Segment;
14 import mgtaylor.layout.*;
15 import mgtaylor.layout.event.GraphLayoutEvent;
16 import mgtaylor.planarity.event.*;

18 public class BlockDataView extends DataView {
19     /**/
20     private static final long serialVersionUID = -6835917939703146492L;

22     private final BlockTableModel model;

```

```

24 // private final JComponentTableRenderer renderer = new
JComponentTableRenderer();
25 public BlockDataView( final VertexLayout layout, final PlanarityHandler
planarity ) {
26     super(layout);

28     final UnmodifiableLinkList<Block> blocks =
layout.getGraph().getAllBlocks();

30     final Block[] data = new Block[ blocks.size() ];
31     int i = 0;
32     for ( Block block: blocks ) {
33         data[ i++ ] = block;
34     }
35     model = new BlockTableModel( data );
36     JTable table = new JTable( model );
37     JScrollPane scroll = new JScrollPane( table );
38     table.setFillViewportHeight( true );

40     if ( table.getColumnCount() > 0 ) {
41         JTableButtonListener flipListener = new JTableButtonListener() {
42             public void tableButtonPressed(JTableButtonEvent event) {
43                 Block block = (Block) model.getValueAt(event.getRow(), 0);
44                 if ( block.isFlippable() ) {
45                     block.invert();
46                     model.fireTableDataChanged();
47                     layout.getGraph().generateEdgeSides();
48                     planarity.processPlanarOrderEvent( new PlanarOrderEvent(
planarity ) );
49                 }
50             }
51         };

53         ButtonColumn flipColumn = new ButtonColumn( table, 1 );
54         flipColumn.addTableButtonListener( flipListener );
55     }

57     setLayout( new BorderLayout() );
58     add( scroll, BorderLayout.CENTER );
59 }

62 @Override
63 public void handleGraphLayoutChanged( final GraphLayoutEvent event ) {}

65 public static class BlockTableModel extends AbstractTableModel {
66     /**/
67     private static final long serialVersionUID = -8006176331578385316L;
68     final Class<?>[] types = new Class<?>[] {
69         Block.class,
70         String.class,
71         Boolean.class,
72         Boolean.class,
73         Segment.class,
74         String.class
75     };
76     final String[] headers = new String[] {
77         "Block",
78         "Flip",
79         "Can Flip?",
80         "Is Flipped?",
81         "Parent",
82         "Status"
83     };

```

```

85     private final Block[] data;

87     public BlockTableModel( Block[] data ) {
88         this.data = data;
89     }

91     public String getColumnName(int col) { return headers[col]; }

93     public int getRowCount() { return data.length; }
94     public int getColumnCount() {
95         if ( data != null && data.length > 0 )
96             return headers.length;
97         return 0;
98     }

100    public Object getValueAt(int row, int col) {
101        switch( col ) {
102            case 0: return data[row];
103            case 1: return ">";
104            case 2: return data[row].isFlippable();
105            case 3: return data[row].isInverted();
106            case 4: return data[row].getParent();
107            case 5: return (data[row].isRemoved())?"Removed":"Active";
108        }
109        return null;
110    }

112    public Class<?> getColumnClass(int c) {
113        return types[c];
114    }

116    public boolean isCellEditable(int row, int col) {
117        return col == 1;
118    }
119    public void setValueAt(Object value, int row, int col) {}
120 }
121 }

```

D.5.9 mgtaylor.display.data.ButtonColumn

```

1 package mgtaylor.display.data;

3 import java.awt.Component;
4 import java.awt.event.ActionEvent;
5 import java.awt.event.ActionListener;
6 import java.util.*;

8 import javax.swing.AbstractCellEditor;
9 import javax.swing.JButton;
10 import javax.swing.JTable;
11 import javax.swing.UIManager;
12 import javax.swing.table.TableCellEditor;
13 import javax.swing.table.TableCellRenderer;
14 import javax.swing.table.TableColumnModel;

16 import mgtaylor.display.data.event.*;

19 public class ButtonColumn extends AbstractCellEditor
20 implements TableCellRenderer, TableCellEditor, ActionListener,
    JTableButtonHandler {

```

```

21  /**
22  *
23  */
24  private static final long serialVersionUID = 2985480381256465894L;
25  private final HashSet<JTableButtonListener> listeners = new
    HashSet<JTableButtonListener>();
26  private final JTable table;
27  private final JButton renderButton;
28  private final JButton editButton;
29  private String text = null;

31  public ButtonColumn(JTable table, int column)
32  {
33      super();
34      this.table = table;
35      renderButton = new JButton();

37      editButton = new JButton();
38      editButton.setFocusPainted( false );
39      editButton.addActionListener( this );

41      TableColumnModel columnModel = table.getColumnModel();
42      columnModel.getColumn(column).setCellRenderer( this );
43      columnModel.getColumn(column).setCellEditor( this );
44  }

46  public Component getTableCellRendererComponent(
47      JTable table, Object value, boolean isSelected, boolean hasFocus, int
    row, int column)
48  {
49      if (hasFocus || !isSelected) {
50          renderButton.setForeground(table.getForeground());
51          renderButton.setBackground(UIManager.getColor("Button.background"));
52      } else {
53          renderButton.setForeground(table.getForeground());
54          renderButton.setBackground(table.getSelectionBackground());
55      }

57      renderButton.setText( (value == null) ? "" : value.toString() );
58      return renderButton;
59  }

61  public Component getTableCellEditorComponent(
62      JTable table, Object value, boolean isSelected, int row, int column)
63  {
64      text = (value == null) ? "" : value.toString();
65      editButton.setText( text );
66      return editButton;
67  }

69  public Object getCellEditorValue()
70  {
71      return text;
72  }

74  public void actionPerformed(ActionEvent e) {
75      fireEditingStopped();
76      // System.out.println( e.getActionCommand() + " : " + table.getSelectedRow());
77      fireTableButtonPressed( new JTableButtonEvent( e, table ) );
78  }

80  @Override
81  public void addTableButtonListener(JTableButtonListener listener) {
82      listeners.add(listener);
83  }

```



```

85     @Override
86     public void fireTableButtonPressed(JTableButtonEvent event) {
87         for ( JTableButtonListener listener : listeners )
88             listener.tableButtonPressed( event );
89     }

91     @Override
92     public void removeTableButtonListener(JTableButtonListener listener) {
93         listeners.remove(listener);
94     }
95 }

```

D.5.10 mgtaylor.display.data.DataView

```

1 package mgtaylor.display.data;

3 import javax.swing.*;

5 import mgtaylor.layout.*;
6 import mgtaylor.layout.event.*;

9 public abstract class DataView extends JPanel implements GraphLayoutListener {
10     /**/
11     private static final long serialVersionUID = -4078807160909725515L;

13     protected final VertexLayout layout;

15     public DataView( final VertexLayout layout ) {
16         this.layout = layout;
17         layout.addGraphLayoutListener( this );
18     }

20     protected VertexLayout getGraphLayout() {
21         return layout;
22     }

24     public abstract void handleGraphLayoutChanged( final GraphLayoutEvent event );
25 }

```

D.5.11 mgtaylor.display.data.PermutationDataView

```

1 package mgtaylor.display.data;

3 import java.awt.*;

5 import javax.swing.*;
6 import javax.swing.table.*;

8 import mgtaylor.display.PlanarityHandler;
9 import mgtaylor.display.data.event.*;
10 import mgtaylor.layout.*;
11 import mgtaylor.layout.event.*;
12 import mgtaylor.planarity.event.PlanarOrderEvent;

14 import mgtaylor.graph.Segment;
15 import mgtaylor.graph.SegmentPermutation;
16 import mgtaylor.datastructures.UnmodifiableLinkedList;

```

```

18 public class PermutationDataView extends DataView {
19     /**/
20     private static final long serialVersionUID = -6835917939703146492L;
21
22     private final PermutationTableModel model;
23
24     // private final JComponentTableRenderer renderer = new
25     JComponentTableRenderer();
26     public PermutationDataView( final VertexLayout layout, final PlanarityHandler
27     planarity ) {
28         super(layout);
29
30         final UnmodifiableLinkedList<SegmentPermutation> permutations =
31         layout.getGraph().getPermutations();
32
33         final SegmentPermutation[] data = new
34         SegmentPermutation[permutations.size()];
35         int i = 0;
36         for ( SegmentPermutation permutation: permutations ) {
37             data[ i ] = permutation;
38             i++;
39         }
40         model = new PermutationTableModel( data );
41         JTable table = new JTable( model );
42         JScrollPane scroll = new JScrollPane( table );
43         table.setFillViewportHeight( true );
44
45         if ( table.getColumnCount() > 0 ) {
46             JTableButtonListener resetListener = new JTableButtonListener() {
47                 public void tableButtonPressed(JTableButtonEvent event) {
48                     SegmentPermutation permutation = (SegmentPermutation) data[
49                     event.getRow() ];
50                     permutation.jumpToPermutation( 0 );
51                     model.fireTableDataChanged();
52                     layout.getGraph().generateEdgeSides();
53                     planarity.processPlanarOrderEvent( new PlanarOrderEvent(
54                     planarity ) );
55                 }
56             };
57
58             JTableButtonListener nextListener = new JTableButtonListener() {
59                 public void tableButtonPressed(JTableButtonEvent event) {
60                     SegmentPermutation permutation = (SegmentPermutation) data[
61                     event.getRow() ];
62                     permutation.nextPermutation();
63                     model.fireTableDataChanged();
64                     layout.getGraph().generateEdgeSides();
65                     planarity.processPlanarOrderEvent( new PlanarOrderEvent(
66                     planarity ) );
67                 }
68             };
69
70             ButtonColumn resetColumn = new ButtonColumn( table, 0 );
71             resetColumn.addTableButtonListener( resetListener );
72
73             ButtonColumn nextColumn = new ButtonColumn( table, 2 );
74             nextColumn.addTableButtonListener( nextListener );
75         }
76
77         setLayout( new BorderLayout() );
78         add( scroll, BorderLayout.CENTER );
79     }

```

```

74  @Override
75  public void handleGraphLayoutChanged( final GraphLayoutEvent event ) {}

77  public static class PermutationTableModel extends AbstractTableModel {
78      /**/
79      private static final long serialVersionUID = -8006176331578385316L;
80      final Class<?>[] types = new Class<?>[] {
81          String.class,
82          SegmentPermutation.class,
83          String.class,
84          Boolean.class,
85          Segment.class,
86          Segment.class,
87          Integer.class,
88          Integer.class
89      };
90      final String[] headers = new String[] {
91          "Reset",
92          "Permutation",
93          "Next",
94          "Is Permutable With Parent?",
95          "Parent",
96          "First Child",
97          "Index",
98          "Total Permutations"
99      };

101     private final SegmentPermutation[] data;

103     public PermutationTableModel( SegmentPermutation[] data ) {
104         this.data = data;
105     }

107     public String getColumnName( int col ) { return headers[ col ]; }

109     public int getRowCount() { return data.length; }
110     public int getColumnCount() {
111         if ( data != null && data.length > 0 )
112             return headers.length;
113         return 0;
114     }

116     public Object getValueAt( int row, int col ) {
117         switch( col ) {
118             case 0: return "Reset";
119             case 1: return data[ row ].getPermutation();
120             case 2: return ">";
121             case 3: return data[ row ].isPermutableWithParent();
122             case 4: return data[ row ].getParentSegment();
123             case 5: return null; // data[ row ].getFirstChild();
124             case 6: return data[ row ].getCurrentPermutation();
125             case 7: return data[ row ].getTotalPermutations();
126         }
127         return null;
128     }

130     public Class<?> getColumnClass( int c ) {
131         return types[ c ];
132     }

134     public boolean isCellEditable( int row, int col ) {
135         return col == 0 || col == 2;
136     }
137     public void setValueAt( Object value, int row, int col ) {}
138 }

```

139 }

D.5.12 mgtaylor.display.data.SegmentDataView

```

1 package mgtaylor.display.data;

3 import java.awt.*;
4 import java.util.*;

6 import javax.swing.*;
7 import javax.swing.table.*;

9 import mgtaylor.layout.*;
10 import mgtaylor.layout.event.*;
11 import mgtaylor.planarity.event.PlanarOrderEvent;
12 import mgtaylor.planarity.event.PlanarOrderListener;

14 import mgtaylor.graph.Segment;
15 import mgtaylor.graph.Block;
16 import mgtaylor.datastructures.UnmodifiableLinkedList;
17 import mgtaylor.display.PlanarityHandler;

19 public class SegmentDataView extends DataView implements PlanarOrderListener {
20     /**/
21     private static final long serialVersionUID = -6835917939703146492L;

23     private final SegmentTableModel model;

25     public SegmentDataView( final VertexLayout layout, final PlanarityHandler
planarity ) {
26         super(layout);
27         final UnmodifiableLinkedList<Segment> segments =
layout.getGraph().getSegments();
28         final Segment[] data = new Segment[segments.size()];
29         int i = 0;
30         for ( Segment segment: segments ) {
31             data[ i++ ] = segment;
32         }
33         model = new SegmentTableModel( data );
34         JTable table = new JTable( model );
35         JScrollPane scroll = new JScrollPane( table );
36         table.setFillViewportHeight( true );
37         setLayout( new BorderLayout() );
38         add( scroll, BorderLayout.CENTER );
39         planarity.addPlanarOrderListener( this );
40     }

43     @Override
44     public void handleGraphLayoutChanged( final GraphLayoutEvent event ) {}

46     @Override
47     public void handlePlanarOrderChanged( PlanarOrderEvent event ) {
48         model.fireTableDataChanged();
49     }

51     private class SegmentTableModel extends AbstractTableModel {
52         /**/
53         private static final long serialVersionUID = -8006176331578385316L;
54         final Class<?>[] types = new Class<?>[] {
55             Integer.class,
56             String.class,

```

```

57         LinkedList.class,
58         String.class,
59         Boolean.class,
60         Block.class,
61         Boolean.class,
62         Boolean.class,
63         Boolean.class,
64         String.class,
65         String.class
66     };
67     final String[] headers = new String[] {
68         "ID",
69         "Root",
70         "Stem",
71         "Leaf",
72         "Can Flip?",
73         "Block",
74         "Block Flips?",
75         "Embedded",
76         "Clockwise",
77         "Side",
78         "Conflicts"
79     };
80     private final Segment[] data;
81
82     public SegmentTableModel( Segment[] data ) {
83         this.data = data;
84     }
85
86     public String getColumnName( int col ) { return headers[ col ]; }
87
88     public int getRowCount() { return data.length; }
89     public int getColumnCount() {
90         if ( data != null && data.length > 0 )
91             return headers.length;
92         return 0;
93     }
94
95     public Object getValueAt( int row, int col ) {
96         Segment segment = data[ row ];
97         switch( col ) {
98             case 0: return segment.getSegmentIndex();
99             case 1: return segment.getRootVertex().toString() + " (" +
segment.getRootVertexDFSIndex() + ")";
100             case 2: return segment.getStemVertices();
101             case 3: return segment.getLeafVertex().toString() + " (" +
segment.getLeafVertexDFSIndex() + ")";
102             case 4: return segment.isFlippable();
103             case 5: return segment.getEmbeddedBlock();
104             case 6: return (segment.getEmbeddedBlock() == null?null:new
Boolean(segment.getEmbeddedBlock().isFlippable()));
105             case 7: return segment.isEmbedded();
106             case 8: return new Boolean( segment.getEmbeddingDirection() ==
Segment.CLOCKWISE );
107             case 9: return segment.getSide() == Block.INSIDE?"In":"Out";
108             case 10:
109                 final StringBuffer out = new StringBuffer();
110                 out.append( '<' );
111                 boolean first = true;
112                 for ( final Segment s: segment.getConflictingSegments() ) {
113                     if ( first ) first = false;
114                     else out.append( ", " );
115                     out.append( s.getSegmentIndex() );
116                 }
117                 out.append( '>' );

```

```

118         return out.toString();
119     }
120     return null;
121 }

123 public Class<?> getColumnClass(int c) {
124     return types[c];
125 }

127 public boolean isCellEditable(int row, int col) { return false; }
128 public void setValueAt(Object value, int row, int col) {}
129 }
130 }

```

D.5.13 mgtaylor.display.data.VertexDataView

```

1 package mgtaylor.display.data;

3 import java.awt.BorderLayout;
4 import java.util.*;
5 import javax.swing.*;
6 import javax.swing.table.AbstractTableModel;

8 import mgtaylor.layout.*;
9 import mgtaylor.layout.event.*;
10 import mgtaylor.planarity.event.*;

12 import mgtaylor.datastructures.UnmodifiableLinkedList;
13 import mgtaylor.display.PlanarityHandler;
14 import mgtaylor.graph.*;

16 public class VertexDataView extends DataView implements PlanarOrderListener {
17     /**/
18     private static final long serialVersionUID = -6835917939703146492L;
19     private final Vertex[] data;
20     private final VertexTableModel model;

22     public VertexDataView( final VertexLayout layout, final PlanarityHandler
planarity ) {
23         super(layout);
24         final Graph graph = layout.getGraph();
25         final Collection<Vertex> vertices = graph.getVertices();
26         data = new Vertex[vertices.size()];
27         int i = 0;
28         for ( Vertex vertex: vertices ) {
29             data[ i ] = vertex;
30             i++;
31         }
32         model = new VertexTableModel( data );
33         JTable table = new JTable( model );
34         JScrollPane scroll = new JScrollPane( table );
35         table.setFillViewportHeight( true );
36         setLayout( new BorderLayout() );
37         add( scroll, BorderLayout.CENTER );
38         planarity.addPlanarOrderListener( this );
39     }

41     @Override
42     public void handleGraphLayoutChanged( final GraphLayoutEvent event ) {}

44     @Override
45     public void handlePlanarOrderChanged( PlanarOrderEvent event ) {

```

```

46         model.fireTableDataChanged();
47     }

49     private class VertexTableModel extends AbstractTableModel {
50         /**/
51         private static final long serialVersionUID = -8006176331578385316L;
52         final Class<?>[] types = new Class<?>[] {
53             Vertex.class,
54             ArrayList.class,
55             Integer.class,
56             Integer.class,
57             Integer.class,
58             ArrayList.class,
59             Vertex.class,
60             Segment.class,
61             ArrayList.class
62         };
63         private final String[] headers = new String[] {
64             "Vertex",
65             "Adjacent Vertices",
66             "DFS Index",
67             "Low Pt 1",
68             "Low Pt 2",
69             "DFS Adjacency",
70             "Parent",
71             "Segment",
72             "Planar Order"
73         };

75         private final Vertex[] data;

77         public VertexTableModel( Vertex[] data ) {
78             this.data = data;
79         }

81         public String getColumnName(int col) { return headers[col]; }

83         public int getRowCount() { return data.length; }
84         public int getColumnCount() {
85             if ( data != null && data.length > 0 )
86                 return headers.length;
87             return 0;
88         }

90         public Object getValueAt(int row, int col) {
91             switch( col ) {
92                 case 0: return data[row];
93                 case 1: ArrayList<Edge> edges = data[row].getConnectedEdges();
94                     ArrayList<Vertex> vertices = new ArrayList<Vertex>( edges.size() );
95                     for ( Edge edge: edges )
96                         vertices.add( edge.getOtherEndFrom( data[row] ) );
97                     return vertices;
98                 case 2: return data[row].getDFSIndex();
99                 case 3: return data[row].getLowPoint1();
100                 case 4: return data[row].getLowPoint2();
101                 case 5: UnmodifiableLinkedList<Edge> adjEdges =
102                     data[row].getDFSAdjacency();
103                     ArrayList<Vertex> adjVertices = new ArrayList<Vertex>(
104                         adjEdges.size() );
105                     for ( Edge edge: adjEdges )
106                         adjVertices.add( edge.getOtherEndFrom( data[row] ) );
107                     return adjVertices;
108                 case 6: return data[row].getDFSParent();
109                 case 7: return data[row].getSegment();

```

```

108         case 8: Edge[] ordEdges = data[row].getEdgeOrder();
109             ArrayList<Vertex> ordVertices = new ArrayList<Vertex>(
110                 ordEdges.length );
111             for ( Edge edge: ordEdges )
112                 ordVertices.add( edge.getOtherEndFrom( data[row] ) );
113             return ordVertices;
114         }
115     }
116
117     public Class<?> getColumnClass( int c ) {
118         return types[c];
119     }
120
121     public boolean isCellEditable( int row, int col ) {
122         return false;
123     }
124     public void setValueAt( Object value, int row, int col ) {}
125 }
126 }

```

D.5.14 mgtaylor.display.data.event.JTableButtonEvent

```

1 package mgtaylor.display.data.event;
2
3 import java.awt.event.ActionEvent;
4
5 import javax.swing.*;
6
7 public class JTableButtonEvent extends ActionEvent {
8     /**/
9     private static final long serialVersionUID = 921553754034935026L;
10
11     private final JTable table;
12     private final int row;
13
14     public JTableButtonEvent( ActionEvent event, JTable table ) {
15         super( event.getSource(), event.getID(), event.getActionCommand(),
16             event.getWhen(), event.getModifiers() );
17         this.table = table;
18         this.row = table.getSelectedRow();
19     }
20
21     public int getRow() { return row; }
22     public JTable getTable() { return table; }
23 }

```

D.5.15 mgtaylor.display.data.event.JTableButtonHandler

```

1 package mgtaylor.display.data.event;
2
3 public interface JTableButtonHandler {
4     public void addTableButtonListener( JTableButtonListener listener );
5     public void removeTableButtonListener( JTableButtonListener listener );
6     public void fireTableButtonPressed( JTableButtonEvent event );
7 }

```


D.5.16 mgtaylor.display.data.event.JTableButtonListener

```

1 package mgtaylor.display.data.event;
2
3 public interface JTableButtonListener {
4     public void tableButtonPressed( JTableButtonEvent event );
5 }

```

D.5.17 mgtaylor.display.event.GraphDisplayEvent

```

1 package mgtaylor.display.event;
2
3 import java.awt.*;
4 import java.awt.geom.*;
5 import java.util.*;
6
7 import mgtaylor.graph.*;
8
9 /**
10  *
11  * @author Martyn G Taylor
12  *
13  */
14 public class GraphDisplayEvent extends AWTEvent {
15     /**/
16     private static final long serialVersionUID = -1771309464704592206L;
17     private final Set<Vertex> vertices;
18     private final Point2D.Double offset;
19     private final Map<Vertex, Point2D.Double> offsetMap;
20
21     public GraphDisplayEvent(Graph graph, Set<Vertex> vertices, Point2D.Double
22     offset) {
23         super(graph, 0);
24         this.vertices = vertices;
25         this.offset = offset;
26         this.offsetMap = null;
27     }
28
29     public GraphDisplayEvent(Graph graph, Set<Vertex> vertices, Map<Vertex,
30     Point2D.Double> offsets) {
31         super(graph, 0);
32         this.vertices = vertices;
33         this.offset = null;
34         this.offsetMap = offsets;
35     }
36
37     public Set<Vertex> getVertices() {
38         return vertices;
39     }
40
41     public Point2D.Double getOffset(Vertex vertex) {
42         if (offset != null)
43             return offset;
44         if (offsetMap != null)
45             return offsetMap.get(vertex);
46         return null;
47     }
48 }

```

D.5.18 mgtaylor.display.event.GraphDisplayHandler

```

1 package mgtaylor.display.event;
2
3 public interface GraphDisplayHandler {
4     public void addGraphDisplayListener( GraphDisplayListener listener );
5     public void removeGraphDisplayListener( GraphDisplayListener listener );
6     public void processGraphDisplayEvent( GraphDisplayEvent event );
7 }

```

D.5.19 mgtaylor.display.event.GraphDisplayListener

```

1 package mgtaylor.display.event;
2
3 import java.util.EventListener;
4
5 public interface GraphDisplayListener extends EventListener {
6     public void handleGraphDisplayChanged( GraphDisplayEvent event );
7 }

```

D.5.20 mgtaylor.layout.ConstantMovementMap

```

1 package mgtaylor.layout;
2
3 import java.awt.geom.*;
4 import java.util.*;
5
6 import mgtaylor.graph.Vertex;
7
8 public class ConstantMovementMap implements MovementMap {
9     public final VertexLayout layout;
10    public final HashSet<Vertex> vertices;
11    public final Point2D.Double offset;
12
13    public ConstantMovementMap(VertexLayout layout, HashSet<Vertex> vertices,
14    Point2D.Double offset) {
15        this.layout = layout;
16        this.vertices = vertices;
17        this.offset = offset;
18    }
19
20    @Override
21    public VertexLayout getLayout() {
22        return layout;
23    }
24
25    @Override
26    public Point2D.Double getVertexOffset(Vertex vertex) {
27        return offset;
28    }
29
30    @Override
31    public Point2D.Double getInvertedVertexOffset(Vertex vertex) {
32        return new Point2D.Double( -offset.x, -offset.y );
33    }
34
35    @Override
36    public Set<Vertex> getVertices() {

```

```

36         return vertices;
37     }

39     @Override
40     public boolean contains(Vertex vertex) {
41         return vertices.contains( vertex );
42     }
44 }

```

D.5.21 mgtaylor.layout.MovementMap

```

1 package mgtaylor.layout;

3 import java.awt.geom.*;
4 import java.util.*;

6 import mgtaylor.graph.*;

8 public interface MovementMap {
9     public VertexLayout getLayout();
10    public Set<Vertex> getVertices();
11    public boolean contains( Vertex vertex );
12    public Point2D.Double getVertexOffset( Vertex vertex );
13    public Point2D.Double getInvertedVertexOffset( Vertex vertex );
14 }

```

D.5.22 mgtaylor.layout.OrderedVertexLayout

```

1 package mgtaylor.layout;

3 import java.awt.geom.Point2D;
4 import java.util.ArrayList;

6 import mgtaylor.graph.Graph;
7 import mgtaylor.graph.Vertex;
8 import mgtaylor.graph.Edge;

10 public class OrderedVertexLayout extends RandomVertexLayout {

12     public OrderedVertexLayout(Graph graph) {
13         super(graph);
14     }

16     @Override
17     public void performLayout() {
18         ArrayList<Vertex> vertices = getGraph().getVertices();
19         Point2D.Double l1 = getVertexLocation( vertices.get( 0 ) );
20         l1.x = 50; l1.y = 95;
21         Point2D.Double l2 = getVertexLocation( vertices.get( 1 ) );
22         l2.x = 95; l2.y = 5;
23         Point2D.Double l3 = getVertexLocation( vertices.get( 2 ) );
24         l3.x = 5; l3.y = 5;
25         for ( int i = 3; i < vertices.size(); i++ ) {
26             Vertex v = vertices.get( i );
27             ArrayList<Edge> edges = v.getConnectedEdges();
28             Point2D.Double p1 = getVertexLocation( edges.get( 0 ).getOtherEndFrom(
                v ) );

```

```

29         Point2D.Double pl2 = getVertexLocation( edges.get( 1 ).getOtherEndFrom(
30             v ) );
31         Point2D.Double pl3 = getVertexLocation( edges.get( 2 ).getOtherEndFrom(
32             v ) );
33         Point2D.Double loc = getVertexLocation( v );
34         loc.x = (pl1.x + pl2.x + pl3.x)/3;
35         loc.y = (pl1.y + pl2.y + pl3.y)/3;
36     }
}

```

D.5.23 mgtaylor.layout.PermutableFlippableLayout

```

1 package mgtaylor.layout;

3 import java.awt.geom.Point2D;
4 import java.util.ArrayList;

6 import mgtaylor.graph.Graph;
7 import mgtaylor.graph.Vertex;
8 import mgtaylor.graph.graphTypes.PermutableFlippableGraph;

10 public class PermutableFlippableLayout extends RandomVertexLayout {

12     public PermutableFlippableLayout(Graph graph) {
13         super(graph);
14     }

16     @Override
17     public void performLayout() {
18         if ( this.getGraph() instanceof PermutableFlippableGraph ) {
19             final PermutableFlippableGraph g = (PermuableFlippableGraph)
20                 this.getGraph();
21             final ArrayList<Vertex> vertices = g.getVertices();
22             setLocation( vertices.get(0), 10.0, 50.0 );
23             setLocation( vertices.get(1), 50.0, 10.0 );
24             setLocation( vertices.get(2), 50.0, 90.0 );
25             final int s = vertices.size() - 3;
26             setLocation( vertices.get(3), 90.0, 50.0 );
27             setLocation( vertices.get(4), 90.0 - 80.0 / s, 50.0 );
28             for ( int i = 3; i <= s; i++ )
29                 setLocation( vertices.get(i + 2), (10.0 + (80.0 * (i - 2)) / s ),
30                     50.0 );
31         }
32     }

33     private void setLocation( Vertex vertex, double x, double y ) {
34         Point2D.Double loc = getVertexLocation( vertex );
35         loc.x = x;
36         loc.y = y;
37     }
}

```

D.5.24 mgtaylor.layout.PlanarWheelLayout

```

1 package mgtaylor.layout;

3 import java.awt.geom.Point2D;
4 import java.util.*;

```

```

6 import mgtaylor.graph.*;
7 import mgtaylor.graph.graphTypes.PlanarMultiWheelGraph;
8 import mgtaylor.graph.graphTypes.PlanarWheelGraph;

10 public class PlanarWheelLayout extends RandomVertexLayout {

12     public PlanarWheelLayout(Graph graph) {
13         super(graph);
14     }

16     @Override
17     public void performLayout() {
18         if ( this.getGraph() instanceof PlanarWheelGraph ) {
19             PlanarWheelGraph pwg = (PlanarWheelGraph) this.getGraph();
20             ArrayList<Vertex> rimVertices = pwg.getRimVertices();
21             switch ( rimVertices.size() ) {
22                 case 0:
23                     return;
24                 case 1:
25                     setLocation( rimVertices.get(0), 50.0, 50.0 );
26                     return;
27                 case 2:
28                     setLocation( rimVertices.get(0), 25.0, 50.0 );
29                     setLocation( rimVertices.get(1), 75.0, 50.0 );
30                     return;
31                 default:
32                     for ( int i = 0; i < rimVertices.size(); i++ ) {
33                         double angle = (40.0 * ( 2.0d * i / (rimVertices.size() - 1) ) -
34                             40.0d) * Math.PI / 180.0d;
35                         setLocation( rimVertices.get(i),
36                             50.0 + 70.0d * Math.sin( angle ),
37                             -50 + 70.0d * Math.cos( angle ) );
38                     }
39                     Vertex innerHub = pwg.getInnerHubVertex();
40                     if ( innerHub != null )
41                         setLocation( innerHub, 50.0, 8.0 );
42                     Vertex outerHub = pwg.getOuterHubVertex();
43                     if ( outerHub != null )
44                         setLocation( outerHub, 50.0, 95.0 );
45             } else if ( this.getGraph() instanceof PlanarMultiWheelGraph ) {
46                 PlanarMultiWheelGraph pmwg = (PlanarMultiWheelGraph) this.getGraph();
47                 Vertex center = pmwg.getCenterVertex();
48                 if ( center != null ) {
49                     setLocation( center, 50.0, 50.0 );
50                     ArrayList<Vertex> hubs = pmwg.getHubs();
51                     ArrayList<ArrayList<Vertex>> spokes = pmwg.getSpokes();
52                     final int numberOfSpokes = hubs.size();
53                     double angle = 2.0d * Math.PI / numberOfSpokes;
54                     for ( int i = 0; i < numberOfSpokes; i++ ) {
55                         double hr = 35.0 * Math.cos( angle / 2.0 );
56                         setLocation( hubs.get( i ),
57                             50.0 + hr * Math.cos( angle * i - angle / 2.0 ),
58                             50.0 + hr * Math.sin( angle * i - angle / 2.0 ) );
59                         ArrayList<Vertex> spoke = spokes.get( i );
60                         for ( int s = 0; s < spoke.size(); s++ ) {
61                             double r = 45.0 * ( s + 1 ) / spoke.size();
62                             setLocation( spoke.get( s ),
63                                 50.0 + r * Math.cos( angle * i ),
64                                 50.0 + r * Math.sin( angle * i ) );
65                         }
66                     }
67                 }
68             } else {

```

```

69         super.performLayout();
70     }
71 }

73 private void setLocation( Vertex vertex, double x, double y ) {
74     Point2D.Double loc = getVertexLocation( vertex );
75     loc.x = x;
76     loc.y = y;
77 }
78 }

```

D.5.25 mgtaylor.layout.RandomVertexLayout

```

1 package mgtaylor.layout;
2
3 import java.awt.geom.Point2D;
4
5 import mgtaylor.graph.*;
6
7 public class RandomVertexLayout extends VertexLayout {
8
9     public RandomVertexLayout(Graph graph) {
10         super(graph);
11     }
12
13     @Override
14     public void performLayout() {
15         final int sz = getGraph().vertexSize();
16         for (int i = 0; i < sz; i++) {
17             final Vertex node = getGraph().getVertices().get(i);
18             Point2D.Double loc = getVertexLocation(node);
19             loc.x = 50 + 45 * Math.cos( i * Math.PI * 2 / sz );
20             loc.y = 50 + 45 * Math.sin( i * Math.PI * 2 / sz );
21
22         }
23         /*
24          for (final Vertex node: getGraph().getVertices()) {
25              Point2D.Double loc = getVertexLocation(node);
26              loc.x = Math.random() * 100;
27              loc.y = Math.random() * 100;
28          }
29          */
30     }
31 }

```

D.5.26 mgtaylor.layout.TriangularPrismMaximalPlanarLayout

```

1 package mgtaylor.layout;
2
3 import java.awt.geom.Point2D;
4 import java.util.ArrayList;
5
6 import mgtaylor.graph.*;
7
8 public class TriangularPrismMaximalPlanarLayout extends VertexLayout {
9
10     public TriangularPrismMaximalPlanarLayout(Graph graph) {
11         super(graph);
12     }
13 }

```

```

14  @Override
15  public void performLayout() {
16      final ArrayList<Vertex> vertices = getGraph().getVertices();
17      final boolean finishingExtra = vertices.size() % 3 > 1;
18      final int i;
19      if ( vertices.size() % 3 > 0 ) {
20          Point2D.Double loc = getVertexLocation( vertices.get( 0 ) );
21          loc.x = 50.0d;
22          loc.y = 60.0d;
23          i = 1;
24      } else
25          i = 0;
26      final int sections = vertices.size() / 3;
27      final int ysize = sections + (finishingExtra?1:0);
28      for ( int s = 1; s <= sections; s++ ) {
29          Point2D.Double loc = getVertexLocation( vertices.get( s * 3 - 3 + i ) );
30          loc.x = 50.0d - s * 45.0d / sections;
31          loc.y = 60.0d + s * 35.0d / sections;
32          loc = getVertexLocation( vertices.get( s * 3 - 2 + i ) );
33          loc.x = 50.0d + s * 45.0d / sections;
34          loc.y = 60.0d + s * 35.0d / sections;
35          loc = getVertexLocation( vertices.get( s * 3 - 1 + i ) );
36          loc.x = 50.0d;
37          loc.y = 60.0d - s * 55.0d / ysize;
38      }
39      if ( finishingExtra ) {
40          Point2D.Double loc = getVertexLocation( vertices.get( vertices.size() -
41          1 ) );
42          loc.x = 50.0d;
43          loc.y = 5.0d;
44      }
45  }

```

D.5.27 mgtaylor.layout.VertexLayout

```

1  package mgtaylor.layout;

3  import java.util.*;
4  import java.awt.geom.*;

6  import mgtaylor.layout.event.*;

8  import mgtaylor.datastructures.LinkList;
9  import mgtaylor.graph.*;

11 public abstract class VertexLayout implements GraphLayoutHandler {
12     private final Graph graph;
13     private final VertexLayout previousLayout;
14     private final LinkList<GraphLayoutListener> listeners = new
15     LinkList<GraphLayoutListener>();
16     private final LinkList<MovementMap> history = new LinkList<MovementMap>();
17     private final LinkList<MovementMap> undoHistory = new LinkList<MovementMap>();

18     private final HashMap<Vertex, Point2D.Double> pointMap = new HashMap<Vertex,
19     Point2D.Double>();

20     public VertexLayout(Graph graph) {
21         this.graph = graph;
22         this.previousLayout = null;
23         for (Vertex vertex: graph.getVertices()) {

```

```

24         pointMap.put(vertex, new Point2D.Double(0, 0));
25     }
26     performLayout();
27 }

29 public VertexLayout(VertexLayout previousLayout) {
30     this.graph = previousLayout.getGraph();
31     this.previousLayout = previousLayout;
32     for (Vertex vertex: graph.getVertices()) {
33         final Point2D.Double loc = new Point2D.Double(0, 0);
34         final Point2D.Double prev = previousLayout.getVertexLocation(vertex);
35         loc.x = prev.x;
36         loc.y = prev.y;
37         pointMap.put(vertex, loc);
38     }
39     performLayout();
40 }

42 public Graph getGraph() { return graph; }
43 public VertexLayout getPreviousLayout() { return
previousLayout; }

45 public Point2D.Double getVertexLocation(final Vertex vertex) {
46     return pointMap.get(vertex);
47 }

49 public Point2D.Double getVertexLocation(final Vertex vertex, final
Point2D.Double offset) {
50     final Point2D.Double p = pointMap.get(vertex);
51     if ( offset == null || p == null )
52         return p;
53     double x = Math.min( 100.0, Math.max( 0.0, p.x + offset.x));
54     double y = Math.min( 100.0, Math.max( 0.0, p.y + offset.y));
55     return new Point2D.Double( x, y );
56 }

58 public Point2D.Double getVertexLocation(final Vertex vertex, final
MovementMap map ) {
59     if ( map != null && map.contains( vertex ))
60         return getVertexLocation( vertex, map.getVertexOffset(vertex) );
61     return getVertexLocation( vertex );
62 }

64 public void processMovement( MovementMap map ) {
65     if ( this == map.getLayout() ) {
66         history.addHead(map);
67         undoHistory.clear();
68         moveVerticesBy( map, false );
69     }
70 }

72 public boolean canUndo() { return !history.isEmpty(); }
73 public boolean canRedo() { return !undoHistory.isEmpty(); }

75 public void undo() {
76     if ( canUndo() ) {
77         final MovementMap map = history.removeHead();
78         undoHistory.addHead( map );
79         moveVerticesBy( map, true );
80     }
81 }

83 public void redo() {
84     if ( canRedo() ) {
85         final MovementMap map = undoHistory.removeHead();

```



```

86         history.addHead( map );
87         moveVerticesBy( map, false );
88     }
89 }

91 protected void moveVerticesBy( MovementMap map, boolean invert ) {
92     if ( this == map.getLayout() ) {
93         for ( Vertex vertex: map.getVertices() )
94             if ( pointMap.containsKey( vertex ) ) {
95                 if ( invert )
96                     pointMap.put( vertex, getVertexLocation( vertex,
97                         map.getInvertedVertexOffset( vertex ) ) );
98                 else
99                     pointMap.put( vertex, getVertexLocation( vertex,
100                         map.getVertexOffset( vertex ) ) );
101             }
102     }
103     processGraphLayoutEvent( new GraphLayoutEvent( this, map ) );
104 }

104 protected void setVertexLocation( final Vertex vertex, final Point2D.Double
point ) {
105     if ( pointMap.containsKey( vertex ) )
106         pointMap.remove( vertex );
107     final Point2D.Double p = new Point2D.Double(
108         Math.min( 100.0, Math.max( 0.0, point.x ) ),
109         Math.min( 100.0, Math.max( 0.0, point.y ) ) );
110     pointMap.put( vertex, p );
111 }

113 public abstract void performLayout();

115 public void addGraphLayoutListener( GraphLayoutListener listener ) {
116     listeners.addTail( listener );
117 }

119 public void removeGraphLayoutListener( GraphLayoutListener listener ) {
120     listeners.remove( listener );
121 }

122 }

124 public void processGraphLayoutEvent( GraphLayoutEvent event ) {
125     for ( GraphLayoutListener listener: listeners )
126         listener.handleGraphLayoutChanged( event );
127 }
128 }

```

D.5.28 mgtaylor.layout.event.GraphLayoutEvent

```

1 package mgtaylor.layout.event;
2
3 import java.awt.*;
4
5 import mgtaylor.layout.*;
6
7 /**
8  *
9  * @author Martyn G Taylor
10  *
11  */
12
13 public class GraphLayoutEvent extends AWTEvent {

```

```

14  /***/
15  private static final long serialVersionUID = -1771309464704592206L;
16  private final VertexLayout layout;
17  private final MovementMap map;

19  public GraphLayoutEvent(VertexLayout layout, MovementMap map) {
20      super(layout, 0);
21      this.layout = layout;
22      this.map = map;
23  }

25  public VertexLayout getLayout() {
26      return layout;
27  }
28  public MovementMap getMovementMap() {
29      return map;
30  }
31  }

```

D.5.29 mgtaylor.layout.event.GraphLayoutHandler

```

1  package mgtaylor.layout.event;

3  public interface GraphLayoutHandler {
4      public void addGraphLayoutListener( GraphLayoutListener listener );
5      public void removeGraphLayoutListener( GraphLayoutListener listener );
6      public void processGraphLayoutEvent( GraphLayoutEvent event );
7  }

```

D.5.30 mgtaylor.layout.event.GraphLayoutListener

```

1  package mgtaylor.layout.event;

3  import java.util.EventListener;

5  public interface GraphLayoutListener extends EventListener {
6      public void handleGraphLayoutChanged( final GraphLayoutEvent event );
7  }

```

D.5.31 mgtaylor.planarity.event.PlanarOrderEvent

```

1  package mgtaylor.planarity.event;

3  import java.awt.*;

5  import mgtaylor.display.PlanarityHandler;

7  import mgtaylor.graph.*;

9  public class PlanarOrderEvent extends AWTEvent {
10     /***/
11     private static final long serialVersionUID = -1002995001888288200L;

13     public PlanarOrderEvent(PlanarityHandler test) {
14         super(test, 0);

```

```

15     }
17     public PlanarityHandler getPlanarity() {
18         return (PlanarityHandler) getSource();
19     }
21     public Edge[] getOrder( Vertex vertex ) {
22         return vertex.getEdgeOrder();
23     }
24 }

```

D.5.32 mgtaylor.planarity.event.PlanarOrderHandler

```

1 package mgtaylor.planarity.event;
3 public interface PlanarOrderHandler {
4     public void addPlanarOrderListener( PlanarOrderListener listener );
5     public void removePlanarOrderListener( PlanarOrderListener listener );
6     public void processPlanarOrderEvent( PlanarOrderEvent event );
8 }

```

D.5.33 mgtaylor.planarity.event.PlanarOrderListener

```

1 package mgtaylor.planarity.event;
3 public interface PlanarOrderListener {
4     public void handlePlanarOrderChanged( PlanarOrderEvent event );
5 }

```

D.5.34 mgtaylor.tools.Trigonometry

```

1 package mgtaylor.tools;
3 /**
4  * @author Martyn Taylor
5  * @since 2004-11-15
6  */
7 public final class Trigonometry {
8     /**
9      * Array to store cosine lookup values.
10     */
11     * @see Trigonometry.cos
12     */
13     private static final double[] COSANGLES = new double[91];
15     /**
16     * Array to store cosine lookup values.
17     */
18     * @see Trigonometry.sin
19     */
20     private static final double[] SINANGLES = new double[91];
22     /**
23     * Array to store cosine lookup values.

```

```

24      *
25      * @see Trigonometry.tan
26      */
27      private static final double[] TANANGLES = new double[91];

29      /**
30      * Array to store cosine lookup values.
31      *
32      * @see Trigonometry.arccos
33      */
34      private static final double[] ARCCOSANGLES = new double[90];

36      /**
37      * Array to store cosine lookup values.
38      *
39      * @see Trigonometry.arcsin
40      */
41      private static final double[] ARCSINANGLES = new double[90];

43      /**
44      * Array to store cosine lookup values.
45      *
46      * @see Trigonometry.arctan
47      */
48      private static final double[] ARCTANANGLES = new double[90];

50      /**
51      * Static block to initialise lookup arrays defined above.
52      */
53      static {
54          for ( int i = 0; i <= 90; i++ ) {
55              COSANGLES[i] = Math.cos( Math.toRadians( i ) );
56              SINANGLES[i] = Math.sin( Math.toRadians( i ) );
57              TANANGLES[i] = Math.tan( Math.toRadians( i ) );
58          }
59          for ( int i = 0; i < 90; i++ ) {
60              ARCCOSANGLES[i] = Math.cos( Math.toRadians( i + 0.5d ) );
61              ARCSINANGLES[i] = Math.sin( Math.toRadians( i + 0.5d ) );
62              ARCTANANGLES[i] = Math.tan( Math.toRadians( i + 0.5d ) );
63          }
64      }

66      /**
67      * Search through the lookup table to find the closest integer
68      * value of arccos(cos) and returns the associated angle.
69      *
70      * arccos(cos) takes arguments between -1 and 1 and returns
71      * angles between 0° and 180° and an invalid input will return
72      * -1.
73      *
74      * @param cos double
75      * @return int
76      */
77      public static final int arccos(double cos) {
78          if ( cos > 1 || cos < -1 )
79              return -1;

81          double c = Math.abs(cos);
82          if ( c >= ARCCOSANGLES[0] )
83              if ( cos < 0 ) {
84                  return 180;
85              } else {
86                  return 0;
87              }
88          int min = 0;

```

```

89     int max = 90;
90     while (min + 1 < max) {
91         int mid = (min + max) >>> 1;
92         if (c >= ARCCOSANGLES[mid]) max = mid;
93         else min = mid;
94     }
95     if (cos < 0) {
96         return 180 - max;
97     } else {
98         return max;
99     }
100 }

102 /*
103  * arccos(x, y) calls arccos(cos) using cos = y/sqrt(x2 + y2)
104  * and then determines the angle based on values of x and y.
105  *
106  * arccos(x, y) returns values between 0° and 359° where 0°
107  * is vertical and increments clockwise.
108  */
109 /*
110  public static final int arccos(double x, double y) {
111      double r = Math.sqrt(x*x + y*y);
112      if (r == 0)
113          return 0;

115      int angle = arccos(y/r);

117      if (x < 0) {
118          return (360 - angle)%360;
119      } else {
120          return angle;
121      }
122  }
123  /**/

125 /**
126  * Search through the lookup table to find the closest integer
127  * value of arcsin(sin) and returns the associated angle.
128  *
129  * arcsin(sin) takes arguments between -1 and 1 and returns
130  * angles between -90° and 90° and an invalid input will return
131  * -100.
132  *
133  * @param sin double
134  * @return int
135  */
136  public static final int arcsin(double sin) {
137      if (sin > 1 || sin < -1)
138          return -100;

140      double s = Math.abs(sin);
141      if (s < ARCSINANGLES[0])
142          return 0;
143      int min = 0;
144      int max = 90;
145      while (min + 1 < max) {
146          int mid = (min + max) >>> 1;
147          if (s < ARCSINANGLES[mid]) max = mid;
148          else min = mid;
149      }
150      if (sin < 0) {
151          return -max;
152      } else {
153          return max;

```

```

154     }
155 }

157 /*
158  * arcsin(x, y) calls arcsin(sin) using sin = x/sqrt(x2 + y2)
159  * and then determines the angle based on values of x and y.
160  *
161  * arcsin(x, y) returns values between 0° and 359° where 0°
162  * is vertical and increments clockwise.
163  */
164 /*
165  public static final int arcsin(double x, double y) {
166      double r = Math.sqrt(x*x + y*y);
167      if (r == 0)
168          return 0;
169      int angle = arcsin(x/r);

171      if (y < 0) {
172          angle = 180 - angle;
173      } else if (angle < 0) {
174          angle += 360;
175      }
176      return angle;
177  }
178  /**/

180 /**
181  * Search through the lookup table to find the closest integer
182  * value of arctan(tan) and return the associated angle.
183  *
184  * arctan(tan) returns values between ±90°.
185  *
186  * @param tan double
187  * @return int
188  */
189  public static final int arctan(double tan) {
190      double t = Math.abs(tan);
191      if (t < ARCTANANGLES[0])
192          return 0;
193      int min = 0;
194      int max = 90;
195      while (min + 1 < max) {
196          int mid = (min + max) >>> 1;
197          if (t < ARCTANANGLES[mid]) max = mid;
198          else min = mid;
199      }
200      if (tan < 0)
201          return -max;
202      else
203          return max;
204  }

206 /**
207  * arctan(x, y) calls arctan(tan) using tan = y/x
208  * and then determines the angle based on values of x and y.
209  *
210  * arctan(x, y) returns values between 0° and 359° where 0°
211  * is vertical and increments clockwise.
212  *
213  * @param x double
214  * @param y double
215  * @return int
216  */
217  public static final int arctan(double x, double y) {
218      if (x == 0) {

```

```

219         if (y < 0)         return 180;
220                           return 0;
221     }
222     if (y == 0) {
223         if (x > 0)         return 90;
224                           return 270;
225     }
226     int tan = arctan(x/y);
227     if (y < 0) {
228         return tan + 180;
229     } else if (tan < 0) {
230         return tan + 360;
231     } else {
232         return tan;
233     }
234     /*
235     double tan = Math.abs(x/y);
236     int min = 0, max = 90;
237     while (min + 1 < max) {
238         int mid = (min + max) >> 1;
239         if (tan < ARCTANANGLES[mid])    max = mid;
240         else                            min = mid;
241     }
242     if ((min == 0) && (tan < ARCTANANGLES[0]))    max = 0;
243     return (x>0)?(y>0?max:180-max):((y>0)?(max==0?0:360-max):(180+max));
244     */
245 }
246 /**
247  * Returns the value of cosine from the lookup table.
248  *
249  * @param angle int
250  * @return double
251  */
252 public static final double cos(int angle) {
253     if (angle >= 0) {
254         angle = angle % 360;
255     } else {
256         angle = (angle % 360) + 360;
257     }
258     if (angle <= 90)    return  COSANGLES[angle];
259     if (angle <= 180)   return -COSANGLES[180-angle];
260     if (angle <= 270)   return -COSANGLES[angle-180];
261                       return  COSANGLES[360-angle];
262 }
263 /**
264  * Returns the value of sine from the lookup table.
265  *
266  * @param angle int
267  * @return double
268  */
269 public static final double sin(int angle) {
270     if (angle >= 0) {
271         angle = angle % 360;
272     } else {
273         angle = (angle % 360) + 360;
274     }
275     if (angle <= 90)    return  SINANGLES[angle];
276     if (angle <= 180)   return  SINANGLES[180-angle];
277     if (angle <= 270)   return -SINANGLES[angle-180];
278                       return -SINANGLES[360-angle];
279 }
280 /**
281  * Returns the value of tangent from the lookup table.
282  *
283  * @param angle int

```

```
284  * @return double
285  */
286  public static final double tan(int angle) {
287      if (angle >= 0) {
288          angle = angle % 180;
289      } else {
290          angle = (angle % 180) + 180;
291      }
292      if (angle <= 90) return TANANGLES[angle];
293                      return -TANANGLES[180-angle];
294  }
295 }
```


D.6 L^AT_EX Segment Hierarchy Generation

D.6.1 mgtaylor.latex.SegmentHierarchies

```

1 package mgtaylor.latex;

3 import mgtaylor.datastructures.UnmodifiableLinkedList;
4 import mgtaylor.graph.Graph;
5 import mgtaylor.graph.Segment;

7 public class SegmentHierarchies {

9     /**
10     * @param args
11     */
12     public static void main(String[] args) {
13         Graph g;
14         StringBuffer structure;

16         int minSize;
17         structure = new StringBuffer();
18         structure.append( "\\begin{tabularx}{\\textwidth}{ccc}\\n" );
19         minSize = 10;
20         for (int i = 0; i < 3; i++) {
21             structure.append(
22                 "\\t\\setlength{\\unit}{1.2\\parskip}\\n\\t\\begin{tikzpicture}\\n");
23             g = new mgtaylor.graph.graphTypes.PlanarMultiWheelGraph( minSize + 3 *
24                 i, 3, true );
25             g.DFS( g.getVertices().get(0) );
26             g.generateSegments();
27             UnmodifiableLinkedList<Segment> ss = g.getSegments();
28             java.util.HashMap<Segment, java.lang.Character> sm = new
29                 java.util.HashMap<Segment, java.lang.Character>();
30             char c = 'A';
31             for ( Segment s: ss ) {
32                 sm.put(s, new java.lang.Character(c) );
33                 if (c == 'Z')
34                     c = 'a';
35                 else
36                     c++;
37             }
38             ss.getHead().toLaTeXString(structure, sm, 0, 0, false);
39             structure.append( "\\t\\end{tikzpicture}\\n" );
40             if ( i != 2 )
41                 structure.append( "&\\n" );
42             else
43                 structure.append( "\\tabularnewline\\n" );
44         }
45         for (int i = 0; i < 3; i++) {
46             structure.append( "\\t(" );
47             structure.append( (char) ('a' + i) );
48             structure.append( ") " );
49             structure.append( minSize + 3 * i );
50             structure.append( " Vertices\\n" );
51             if ( i != 2 )
52                 structure.append( "&\\n" );
53             else
54                 structure.append( "\\tabularnewline\\n" );
55         }
56         structure.append("\\end{tabularx}\\n");
57         System.out.println( structure );

58         structure = new StringBuffer();

```

```

57     structure.append( "\\begin{tabularx}{\\textwidth}{ccc}\\n" );
58     minSize = 10;
59     for (int i = 0; i < 3; i++) {
60         structure.append(
61             "\\t\\setlength{\\unit}{1.2\\parskip}\\n\\t\\begin{tikzpicture}\\n");
62         g = new mgtaylor.graph.graphTypes.PlanarMultiWheelGraph( minSize + 3 *
63             i, 3, true );
64         g.DFS( g.getVertices().get(g.getVertices().size()-1) );
65         g.generateSegments();
66         UnmodifiableLinkedList<Segment> ss = g.getSegments();
67         java.util.HashMap<Segment, java.lang.Character> sm = new
68             java.util.HashMap<Segment, java.lang.Character>();
69         char c = 'A';
70         for ( Segment s: ss ) {
71             sm.put(s, new java.lang.Character(c) );
72             if (c == 'Z')
73                 c = 'a';
74             else
75                 c++;
76         }
77         ss.getHead().toLaTeXString(structure, sm, 0, 0, false);
78         structure.append( "\\t\\end{tikzpicture}\\n" );
79         if ( i != 2 )
80             structure.append( "&\\n" );
81         else
82             structure.append( "\\tabularnewline\\n" );
83     }
84     for (int i = 0; i < 3; i++) {
85         structure.append( "\\t(" );
86         structure.append( (char) ('a' + i) );
87         structure.append( ") " );
88         structure.append( minSize + 3 * i );
89         structure.append( " Vertices\\n" );
90         if ( i != 2 )
91             structure.append( "&\\n" );
92         else
93             structure.append( "\\tabularnewline\\n" );
94     }
95     structure.append( "\\end{tabularx}\\n" );
96     System.out.println( structure );
97
98     structure = new StringBuffer();
99     structure.append( "\\begin{tabularx}{\\textwidth}{ccc}\\n" );
100     minSize = 9;
101     for (int i = 0; i < 3; i++) {
102         structure.append(
103             "\\t\\setlength{\\unit}{1.2\\parskip}\\n\\t\\begin{tikzpicture}\\n");
104         g = new
105             mgtaylor.graph.graphTypes.TriangularPrismMaximalPlanarGraph( minSize + 3
106                 * i );
107         g.DFS( g.getVertices().get(0) );
108         g.generateSegments();
109         UnmodifiableLinkedList<Segment> ss = g.getSegments();
110         java.util.HashMap<Segment, java.lang.Character> sm = new
111             java.util.HashMap<Segment, java.lang.Character>();
112         char c = 'A';
113         for ( Segment s: ss ) {
114             sm.put(s, new java.lang.Character(c) );
115             if (c == 'Z')
116                 c = 'a';
117             else
118                 c++;
119         }
120         ss.getHead().toLaTeXString(structure, sm, 0, 0, false);
121         structure.append( "\\t\\end{tikzpicture}\\n" );

```

```

115         if ( i != 2 )
116             structure.append( "&\n" );
117         else
118             structure.append( "\\tabularnewline\n" );
119     }
120     for (int i = 0; i < 3; i++ ) {
121         structure.append( "\t(" );
122         structure.append( (char) ('a' + i) );
123         structure.append( ") " );
124         structure.append( minSize + 3 * i );
125         structure.append( " Vertices\n" );
126         if ( i != 2 )
127             structure.append( "&\n" );
128         else
129             structure.append( "\\tabularnewline\n" );
130     }
131     structure.append("\\end{tabularx}\n");
132     System.out.println( structure );

134     structure = new StringBuffer();
135     structure.append( "\\begin{tabularx}{\\textwidth}{ccc}\n" );
136     minSize = 9;
137     for (int i = 0; i < 3; i++ ) {
138         structure.append(
139             "\t\\setlength{\\unit}{1.2\\parskip}\n\t\\begin{tikzpicture}\n");
140         g = new
141             mgtaylor.graph.graphTypes.TriangularPrismMaximalPlanarGraph( minSize + 3
142             * i );
143         g.DFS( g.getVertices().get(g.getVertices().size() - 1) );
144         g.generateSegments();
145         UnmodifiableLinkedList<Segment> ss = g.getSegments();
146         java.util.HashMap<Segment, java.lang.Character> sm = new
147             java.util.HashMap<Segment, java.lang.Character>();
148         char c = 'A';
149         for ( Segment s: ss ) {
150             sm.put(s, new java.lang.Character(c) );
151             if ( c == 'Z' )
152                 c = 'a';
153             else
154                 c++;
155         }
156         ss.getHead().toLaTeXString(structure, sm, 0, 0, false);
157         structure.append( "\t\\end{tikzpicture}\n" );
158         if ( i != 2 )
159             structure.append( "&\n" );
160         else
161             structure.append( "\\tabularnewline\n" );
162     }
163     for (int i = 0; i < 3; i++ ) {
164         structure.append( "\t(" );
165         structure.append( (char) ('a' + i) );
166         structure.append( ") " );
167         structure.append( minSize + 3 * i );
168         structure.append( " Vertices\n" );
169         if ( i != 2 )
170             structure.append( "&\n" );
171         else
172             structure.append( "\\tabularnewline\n" );
173     }
174     structure.append("\\end{tabularx}\n");
175     System.out.println( structure );

```

Appendix E

Matlab

E.1 Matlab Code to Load Data

Listing E.1.1: Loading Code Warm-Up Data (see section 6.3.2)

```
1 data.warmup.triangular = loadGraphData( [],  
    'CodeWarmUp-Triangle\DifferentRoot-Triangle-', { '50', '250', '250a', '250b',  
    '500', '500a', '750', '750a', '1000', '1000a', '1250', '1250a', '1500' }, 'root'  
    );  
2 data.warmup.wheel = loadGraphData( [], 'CodeWarmUp-Wheel\DifferentRoot-', {  
    '250', '500', '750', '1000', '1250', '1500', '2000', '2500' }, 'root' );
```

Listing E.1.2: Loading Different Roots Data

```
1 data.roots.triangular = loadAndSplitGraphData( [],  
    'DifferentRoots-Triangle\DifferentRoot-Triangle-', '250,750,1001,1002', 'root',  
    'vertex', { 250, 750, 1001, 1002 }, { '250', '750', '1001', '1002' } );  
2 data.roots.triangular = loadGraphData( data.roots.triangular,  
    'DifferentRoots-Triangle\DifferentRoot-Triangle-', { '500', '1000', '1250',  
    '1500' }, 'root' );  
  
4 data.roots.wheel = loadAndSplitGraphData( [],  
    'DifferentRoots-Wheel\DifferentRoot-Wheel-', '250,500,750,1000', 'root',  
    'vertex', { 250, 500, 750, 1000 }, { '250', '500', '750', '1000' } );  
5 data.roots.wheel = loadGraphData( data.roots.wheel,  
    'DifferentRoots-Wheel\DifferentRoot-Wheel-', { '1250', '1500' }, 'root' );  
  
7 data.roots.ordered = loadAndSplitGraphData( [],  
    'DifferentRoots-Ordered\DifferentRoot-Ordered-', '250,500,750,1000,1250',  
    'root', 'vertex', { 250, 500, 750, 1000, 1250 }, { '250', '500', '750', '1000',  
    '1250' } );  
8 data.roots.ordered = loadAndSplitGraphData( data.roots.ordered,  
    'DifferentRoots-Ordered\DifferentRoot-Ordered-', '1500,2000', 'root', 'vertex',  
    { 1500, 2000 }, { '1500', '2000' } );  
  
10 data.roots.random = loadAndSplitGraphData( [], 'DifferentRoots-Random\Random-',  
    '250,500,750,1000', 'root', 'vertex', { 250, 500, 750, 1000 }, { '250', '500',  
    '750', '1000' } );  
11 data.roots.random = loadGraphData( data.roots.random,  
    'DifferentRoots-Random\Random-', { '1250', '1500' }, 'root' );  
12 data.roots.random = loadGraphData( [], 'DifferentRoots-Random\Random-', { '250',  
    '500', '750', '1000', '1250', '1500' }, 'root' );
```

Listing E.1.3: Loading Different Sizes (42-4500 Vertices) Data

```

1 data.sizes.ordered = loadGraphData( [], 'DifferentSizes\Ordered-42-4500-', {
  '0', 'N', 'N4', 'N2', '3N4' }, 'vertex' );
2 data.sizes.random = loadGraphData( [], 'DifferentSizes\Random-42-4500-', {
  '0', 'N', 'N4', 'N2', '3N4' }, 'vertex' );
3 data.sizes.triangular = loadGraphData( [],
  'DifferentSizes\Triangular-42-4500-', { '0', 'N', 'N4', 'N2', '3N4' }, 'vertex'
  );
4 data.sizes.wheel = loadGraphData( [], 'DifferentSizes\Wheel-40-4500-', {
  '0', 'N', 'N4', 'N2', '3N4' }, 'vertex' );

```

Listing E.1.4: Loading Different Sizes (500-15000 Vertices) Data

```

1 data.sizes.ordered = loadGraphData( [], 'DifferentSizes\Ordered-501-15000-', {
  '0', 'N', 'N4', 'N2', '3N4' }, 'vertex' );
2 data.sizes.random = loadGraphData( [], 'DifferentSizes\Random-502-15000-', {
  '0', 'N', 'N4', 'N2', '3N4' }, 'vertex' );
3 data.sizes.triangular = loadGraphData( [],
  'DifferentSizes\Triangular-501-15000-', { '0', 'N', 'N4', 'N2', '3N4' },
  'vertex' );
4 data.sizes.wheel = loadGraphData( [], 'DifferentSizes\Wheel-502-15000-', {
  '0', 'N', 'N4', 'N2', '3N4' }, 'vertex' );

```

Listing E.1.5: Loading Different Edges Data

```

1 data.edges.triangular = loadGraphData( [],
  'DifferentEdges\Biconnected\Triangular-', { '0', 'N4', 'N4a', 'N2', '3N4', 'N'
  }, 'edge' );
2 data.edges.wheel = loadGraphData( [], 'DifferentEdges\Biconnected\Wheel-', {
  '0', 'N4', 'N2', '3N4', 'N' }, 'edge' );
3 data.edges.ordered = loadGraphData( [], 'DifferentEdges\Biconnected\Ordered-', {
  '0', 'N4', 'N4a', 'N2', '3N4', 'N' }, 'edge' );
4 data.edges.random = loadGraphData( [], 'DifferentEdges\Biconnected\Random-', {
  '0', 'N4', 'N2', '3N4', 'N' }, 'edge' );

```

E.2 Matlab Code To Generate Figures

Figure	Matlab Code
6.3.1	<pre> 1 PlotShowCodeWarmUp(data.warmup.triangular.s1250, ... 2 '1250 Vertex Triangular Graph', ... 3 'showbetween', [1 1], 'ypercentile', 98, ... 4 'usesubplot', true); </pre>
6.3.2	<pre> 1 PlotShowCodeWarmUp(data.warmup.triangular.s1250, ... 2 '1250 Vertex Triangular Graph', 'showbetween', [2 2]); </pre>

Figure	Matlab Code
6.3.3	<pre> 1 x1 = normrnd(0,1,50,1); x2 = normrnd(0.1,0.9,100,1); 2 binEdges = [-inf ; sort([x1;x2]) ; inf]; 3 binCounts1 = histc(x1 , binEdges, 1); 4 binCounts2 = histc(x2 , binEdges, 1); 5 sumCounts1 = cumsum(binCounts1)./sum(binCounts1); 6 sumCounts2 = cumsum(binCounts2)./sum(binCounts2); 7 ks = abs(sumCounts1 - sumCounts2); 8 x = [binEdges sumCounts1 sumCounts2]; x(any((ks~=max(ks)), 2), :) = []; x 9 figure; stairs(binEdges, sumCounts1, 'r-'); hold on; stairs(binEdges, sumCounts2, 'b-'); 10 arrow([x(1) x(2)], [x(1) x(3)], 4,90,30,1); 11 arrow([x(1) x(3)], [x(1) x(2)], 4,90,30,1); 12 xlabel('x') 13 ylabel('ECDF(x)') 14 legend('~N(0,1)', '~N(0.1,0.9)', 'Location', 'best') 15 title({'Empirical Cumulative Density Function of Two Randomly', 'Generated Distributions Approximating N(0,1) and N(0.1,0.9)'}); </pre> arrow.m is available from: http://www.mathworks.com/matlabcentral/fileexchange/278
6.3.4	<pre> 1 screenSize = get(0,'ScreenSize'); 2 fig = figure('Position', [0 0 screenSize(3) screenSize(4)], 'PaperUnits', 'inches', 'PaperPosition', [0 0 10 7], 'Color', 'white'); 3 subplot(2, 2, 1), 4 PlotNumberOfElements(data.sizes.wheel.s0, { 'Planar Wheel Graph - Number of', 'Data Points Uninterrupted by Garbage Collection' }, 'showasbar', false, 'newfigure', false) 5 axis([0 15000 70 100]); 6 subplot(2, 2, 2), 7 PlotNumberOfElements(data.sizes.triangular.s0, { 'Triangular Prism Graph - Number of', 'Data Points Uninterrupted by Garbage Collection' }, 'showasbar', false, 'newfigure', false) 8 subplot(2, 2, 3), 9 PlotNumberOfElements(data.sizes.ordered.s0, { 'Ordered Face Trisection Graph - Number of', 'Data Points Uninterrupted by Garbage Collection' }, 'showasbar', false, 'newfigure', false) 10 axis([0 15000 70 100]); 11 subplot(2, 2, 4), 12 PlotNumberOfElements(data.sizes.random.s0, { 'Random Face Trisection Graph - Number of', 'Data Points Uninterrupted by Garbage Collection' }, 'showasbar', false, 'newfigure', false) 13 axis([0 15000 70 100]); 14 print(fig, '-dpng', 'Images/Benchmarking/NumberOfNonGCElements-Size-1stRoot.png', '-r150'); </pre>
6.4.1	<pre> 1 x = (-10:10)'; 2 y = 2*x.^3 - 38*x + 19; 3 y(21) = 0; 4 bf = fitdata2(x, y, { 'x3', {'x3','x2'} }, { 'x3', 'x2', 'x' } }, 'userobustfit', false); 5 figure; hold all; 6 plot(x, y, '+', 'MarkerSize', 3, 'DisplayName', 'Data'); 7 fn = fieldnames(bf.data); 8 for f=1:numel(fn), 9 plot(unique(x), bf.data.(fn{f}), '-', 'DisplayName', bf.equation.(fn{f})); 10 end; 11 legend('show', 'Location', 'best'); 12 bf.rmse 13 bf.mae </pre>

Figure	Matlab Code
6.4.2	<pre> 1 x = (-10:10)'; 2 y = 2*x.^3 - 38*x + 19; 3 y(21) = 0; 4 bf = fitdata2(x, y, { 'x3', {'x3','x2'}, ... 5 { 'x3', 'x2', 'x' } }, 'userobustfit', true); 6 figure; hold all; 7 plot(x, y, '+', 'MarkerSize', 3, 'DisplayName', 'Data'); 8 fn = fieldnames(bf.data); 9 for f=1:numel(fn), 10 plot(unique(x), bf.data.(fn{f}), '-', ... 11 'DisplayName', bf.equation.(fn{f})); 12 end; 13 legend('show', 'Location', 'best'); 14 bf.rmse 15 bf.mae </pre>
A.1.1, A.1.2, A.1.3, 7.4.2, A.1.4, A.1.5, A.1.6, A.1.7, A.1.8, 7.4.3, A.1.9, A.1.10, A.1.11, A.1.12, A.1.13, 7.4.4, A.1.14, A.1.15, A.1.16, A.1.17, A.1.18, A.1.19, 7.4.5, A.1.20 and A.1.21	<pre> 1 g.t.triangular = 'Triangular Prism'; 2 g.t.random = 'Random Face Trisection'; 3 g.t.ordered = 'Ordered Face Trisection'; 4 g.t.wheel = 'Planar Wheel'; 5 g.m.triangular = true; 6 g.m.random = false; 7 g.m.ordered = false; 8 g.m.wheel = false; 9 n = fieldnames(data.roots); 10 for i=1:numel(n), 11 r = fieldnames(data.roots.(n{i})); 12 for j=1:numel(r), 13 plotGraphData(data.roots.(n{i}).(r{j}), [r{j}(2:numel(r{j})) ' 14 Vertex ' g.t.(n{i}) ' Graph'], 'percentile', 98, 15 'percentilesteps', 25, 'showbandsinlegend', true, 'lowcolour', [0 16 0 1], 'highcolour', [0 1 0], 'usehsl', true, 'mapx', g.m.(n{i}), 17 'save', ['DifferentRoots\' upper(n{i}(1)) lower(18 n{i}(2:numel(n{i}))) '-' r{j}(2:numel(r{j}))]); 19 end 20 end </pre>
7.4.6	<pre> 1 plotGraphData(data.roots.random.s1000a, '1000 Vertex Random Face Trisection Graphs', 'percentile', 98, 'percentilesteps', 25, 'showbandsinlegend', false, 'lowcolour', [0 0 1], 'highcolour', [0 1 0], 'usehsl', true, 'showdepthsbestfit', false, 'showbestfits', false); 2 plotGraphData(data.roots.random.s1000, '1000 Vertex Random Face Trisection Graphs', 'percentile', 98, 'percentilesteps', 25, 'showbandsinlegend', false, 'lowcolour', [0 0 0], 'highcolour', [1 0 0], 'usehsl', true, 'showdepthsbestfit', false, 'showbestfits', false, 'newfigure', false); 3 print(gcf, '-dpng', 'DifferentRoots\Random-1000a,1000-Overlay.png', '-r150'); </pre>

Figure	Matlab Code
7.5.1, A.2.1, A.2.2, A.2.3, A.2.4, 7.5.2, A.2.10, A.2.11, A.2.12, A.2.13, A.2.14, 7.5.3, A.2.26, A.2.27, A.2.28, A.2.29, 7.5.4, A.2.35, A.2.36, A.2.37 and A.2.38	<pre> 1 gnm.title.triangular = 'Triangular Prism'; 2 gnm.title.random = 'Random Face Trisection'; 3 gnm.title.ordered = 'Ordered Face Trisection'; 4 gnm.title.wheel = 'Planar Wheel'; 5 gnm.root.s0 = '1st'; 6 gnm.root.sN4 = '1/4Nth'; 7 gnm.root.sN2 = '1/2Nth'; 8 gnm.root.s3N4 = '3/4Nth'; 9 gnm.root.sN = 'Nth'; 10 gn = fieldnames(data.sizes); 11 for i=1:numel(gn), 12 rn = fieldnames(data.sizes.(gn{i})); 13 for j=1:numel(rn), 14 plotGraphData(data.sizes.(gn{i}).(rn{j}), [gnm.title.(gn{i}) ' Graph (Root = ' gnm.root.(rn{j}) ')] , 'percentile', 100, 'percentilesteps', 100, 'showbandsinlegend', false, 'showdepthsbestfit', true, 'save', ['DifferentSizes\' upper(gn{i}(1)) lower(gn{i}(2:numel(gn{i}))) '-Small-' rn{j}(2:numel(rn{j}))]); 15 end 16 end </pre>
A.2.15	<pre> 1 plotGraphData(data.sizes.wheel.s0 , 'Planar Wheel Graph (Root = 1st, Vertices > 750)', 'percentile', 100, 'percentilesteps', 100, 'showbandsinlegend', false, 'showdepthsbestfit', false, 'save', 'DifferentSizes\Wheel-Small-0-750', 'betweenx', [750 4500]); </pre>
A.2.16	<pre> 1 plotGraphData(data.sizes.wheel.sN4 , 'Planar Wheel Graph (Root = 1/4Nth, Vertices > 750)', 'percentile', 100, 'percentilesteps', 100, 'showbandsinlegend', false, 'showdepthsbestfit', false, 'save', 'DifferentSizes\Wheel-Small-N4-750', 'betweenx', [750 4500]); </pre>
A.2.17	<pre> 1 plotGraphData(data.sizes.wheel.sN2 , 'Planar Wheel Graph (Root = 1/2Nth, Vertices > 500)', 'percentile', 100, 'percentilesteps', 100, 'showbandsinlegend', false, 'showdepthsbestfit', false, 'save', 'DifferentSizes\Wheel-Small-N2-500', 'betweenx', [500 4500]); </pre>
A.2.18	<pre> 1 plotGraphData(data.sizes.wheel.s3N4 , 'Planar Wheel Graph (Root = 3/4 Nth, Vertices > 750)', 'percentile', 100, 'percentilesteps', 100, 'showbandsinlegend', false, 'showdepthsbestfit', false, 'save', 'DifferentSizes\Wheel-Small-3N4-750', 'betweenx', [750 4500]); </pre>
A.2.19	<pre> 1 plotGraphData(data.sizes.wheel.s3N4a , 'Planar Wheel Graph (Root = 3/4 Nth, Vertices > 750, 2nd run)', 'percentile', 100, 'percentilesteps', 100, 'showbandsinlegend', false, 'showdepthsbestfit', false, 'save', 'DifferentSizes\Wheel-Small-3N4a-750', 'betweenx', [750 4500]); </pre>
A.2.20	<pre> 1 plotGraphData(data.sizes.wheel.sN , 'Planar Wheel Graph (Root = Nth, Vertices > 750)', 'percentile', 100, 'percentilesteps', 100, 'showbandsinlegend', false, 'showdepthsbestfit', false, 'save', 'DifferentSizes\Wheel-Small-N-750', 'betweenx', [750 4500]); </pre>
A.2.39	<pre> 1 plotGraphData(data.sizes.random.s3N4 , 'Random Face Trisection Graph (Root = 3/4Nth, Vertices > 400)', 'percentile', 100, 'percentilesteps', 100, 'showbandsinlegend', false, 'showdepthsbestfit', false, 'save', 'DifferentSizes\Random-Small-3N4-400', 'betweenx', [400 4500]); </pre>

Figure	Matlab Code
A.2.5, A.2.6, A.2.7, A.2.8, A.2.9, A.2.21, A.2.22, A.2.23, A.2.24, A.2.25, A.2.30, A.2.31, A.2.32, A.2.33, A.2.34, A.2.40, A.2.41, A.2.42, A.2.43 and A.2.44	<pre> 1 gnm.title.triangular = 'Triangular Prism'; 2 gnm.title.random = 'Random Face Trisection'; 3 gnm.title.ordered = 'Ordered Face Trisection'; 4 gnm.title.wheel = 'Planar Wheel'; 5 gnm.root.s0 = '1st'; 6 gnm.root.sN4 = '1/4Nth'; 7 gnm.root.sN2 = '1/2Nth'; 8 gnm.root.s3N4 = '3/4Nth'; 9 gnm.root.sN = 'Nth'; 10 gn = fieldnames(data.sizes); 11 for i=1:numel(gn), 12 rn = fieldnames(data.sizes.(gn{i})); 13 for j=1:numel(rn), 14 plotGraphData(data.sizes.(gn{i}).(rn{j}), [gnm.title.(gn{i}) ' Graph (Root = ' gnm.root.(rn{j}) ')]', 'percentile', 100, 'percentilesteps', 100, 'showbandsinlegend', false, 'showdepthsbestfit', true, 'save', ['DifferentSizes\' upper(gn{i}(1)) lower(gn{i}(2:numel(gn{i}))) '-Large-' rn{j}(2:numel(rn{j}))]); 15 end 16 end </pre>
A.3.1, A.3.2, A.3.3, A.3.4, A.3.5, A.3.6, A.3.7, A.3.8, A.3.9, A.3.10, A.3.11, A.3.12, A.3.13, A.3.14, A.3.15, A.3.16, A.3.17, A.3.18, A.3.19, A.3.20, A.3.21 and A.3.22	<pre> 1 gnm.title.random = 'Random Face Trisection'; 2 gnm.title.ordered = 'Ordered Face Trisection'; 3 gnm.title.triangular = 'Triangular Prism'; 4 gnm.title.wheel = 'Planar Wheel'; 5 gnm.size.random = '14999'; 6 gnm.size.ordered = '14999'; 7 gnm.size.triangular = '14999'; 8 gnm.size.wheel = '15000'; 9 gnm.root.s0 = '1st'; 10 gnm.root.sN4 = '1/4Nth'; 11 gnm.root.sN4a = '1/4Nth, 2nd Run'; 12 gnm.root.sN2 = '1/2Nth'; 13 gnm.root.s3N4 = '3/4Nth'; 14 gnm.root.sN = 'Nth'; 15 gn = fieldnames(data.edges); 16 for i=1:numel(gn), 17 rn = fieldnames(data.edges.(gn{i})); 18 for j=1:numel(rn), 19 plotGraphData(data.edges.(gn{i}).(rn{j}), {'Execution Time of ', [gnm.size.(gn{i}) ' Vertex ' gnm.title.(gn{i}) ' Graph (Root = ' gnm.root.(rn{j}) ')]'}, 'percentile', 100, 'percentilesteps', 100, 'showbandsinlegend', false, 'showdepthsbestfit', false, 'usethresholdlimit', true, 'thresholdlimit', [0 300000000], 'save', ['DifferentEdges\Biconnected\' upper(gn{i}(1)) lower(gn{i}(2:numel(gn{i}))) '- rn{j}(2:numel(rn{j}))]', 'betweenx', [30000 45000]); 20 end 21 end </pre>

Figure	Matlab Code
7.6.2	<pre> 1 screenSize = get(0,'ScreenSize'); fig = figure('Position', [0 0 screenSize(3) screenSize(4)], 'PaperUnits', 'inches', 'PaperPosition', [0 0 10 7], 'Color', 'white'); 2 subplot(1,3,[1 2]), axis([30000 40000 600000000 3000000000]), plotGraphData(data.edges.triangular.sN4, {'Execution Time of 14999 Vertex', 'Triangular Prism Graph (Root = 1/4Nth)'}, 'percentile', 100, 'percentilesteps', 100, 'showbandsinlegend', false, 'showdepthsbestfit', false, 'betweenx', [30000 40000], 'newfigure', false); 3 subplot(1,3,3), axis([40000 45000 600000000 3000000000]), plotGraphData(data.edges.triangular.sN4, {'Execution Time of 14999 Vertex', 'Triangular Prism Graph', '(Root = 1/4Nth)'}, 'percentile', 100, 'percentilesteps', 100, 'showbandsinlegend', false, 'showdepthsbestfit', false, 'newfigure', false, 'save', 'DifferentEdges\Biconnected\Triangular-N4-Split', 'betweenx', [40000 45000]); </pre>
7.6.3	<pre> 1 screenSize = get(0,'ScreenSize'); fig = figure('Position', [0 0 screenSize(3) screenSize(4)], 'PaperUnits', 'inches', 'PaperPosition', [0 0 10 7], 'Color', 'white'); 2 subplot(1,3,[1 2]), axis([30000 40000 600000000 3000000000]), plotGraphData(data.edges.triangular.sN4a, {'Execution Time of 14999 Vertex', 'Triangular Prism Graph (Root = 1/4Nth, 2nd Run)'}, 'percentile', 100, 'percentilesteps', 100, 'showbandsinlegend', false, 'showdepthsbestfit', false, 'betweenx', [30000 40000], 'newfigure', false); 3 subplot(1,3,3), axis([40000 45000 600000000 3000000000]), plotGraphData(data.edges.triangular.sN4a, {'Execution Time of 14999 Vertex', 'Triangular Prism Graph', '(Root = 1/4Nth, 2nd Run)'}, 'percentile', 100, 'percentilesteps', 100, 'showbandsinlegend', false, 'showdepthsbestfit', false, 'newfigure', false, 'save', 'DifferentEdges\Biconnected\Triangular-N4a-Split', 'betweenx', [40000 45000]); </pre>

Figure	Matlab Code
A.4.1, 7.6.4, A.4.2, A.4.3, A.4.4, A.4.5, A.4.6, A.4.7, A.4.8, A.4.9, A.4.10, A.4.11, A.4.12, A.4.13, A.4.14, A.4.15, A.4.16, A.4.17, A.4.18 and A.4.19	<pre> 1 screenSize = get(0,'ScreenSize'); 2 gnm.title.random = 'Random Face Trisection'; 3 gnm.title.ordered = 'Ordered Face Trisection'; 4 gnm.title.triangular = 'Triangular Prism'; 5 gnm.title.wheel = 'Planar Wheel'; 6 gnm.size.random = '14999'; 7 gnm.size.ordered = '14999'; 8 gnm.size.triangular = '14999'; 9 gnm.size.wheel = '15000'; 10 gnm.root.s0 = '1st'; 11 gnm.root.sN4 = '1/4Nth'; 12 gnm.root.sN4a = '1/4Nth, 2nd Run'; 13 gnm.root.sN2 = '1/2Nth'; 14 gnm.root.s3N4 = '3/4Nth'; 15 gnm.root.sN = 'Nth'; 16 gn = fieldnames(data.edges); 17 for i=1:numel(gn), 18 rn = fieldnames(data.edges.(gn{i})); 19 for j=1:numel(rn), 20 d = data.edges.(gn{i}).(rn{j}).data.depths; 21 dd = d(2:size(d,1),2) - d(1:size(d,1)-1,2); 22 ddd = dd(2:size(dd,1),1) - dd(1:size(dd,1)-1,1); 23 fig = figure('Position', [0 0 screenSize(3) screenSize(4)], 24 'PaperUnits', 'inches', 'PaperPosition', [0 0 10 7], 'Color', 25 'white'); 26 t1 = [gnm.size.(gn{i}) ' Vertex ' gnm.title.(gn{i})]; 27 t2 = [' Graph (Root = ' gnm.root.(rn{j}) ' Vertex)']; 28 subplot(2,2,1), hold all, plot(d(:, 6), d(:, 4), 'b'); 29 a = axis; axis([30000 45000 a(3) a(4)]); 30 xlabel('Number Of Edges'); ylabel('Number of Conflicts'); 31 title({'Segment Conflicts for Biconnected Sub-Graphs of', t1, 32 t2 }); 33 subplot(2,2,2), hold all, plot(d(:, 6), d(:, 2), 'b'); 34 a = axis; axis([30000 45000 a(3) a(4)]); 35 xlabel('Number Of Edges'); ylabel('Mean Segment Depth'); 36 title({'Mean Segment Depth for Biconnected Sub-Graphs of', t1, 37 t2 }); 38 subplot(2,2,3), hold all, plot(d(2:size(d,1),6), dd); 39 a = axis; if a(3) < -15, a(3) = -15; end; if a(4) > 20, a(4) = 40 20; end; axis([30000 45000 a(3) a(4)]); 41 title({'Change in Mean Segment Depth for Biconnected 42 Sub-Graphs of', t1, t2 }); 43 xlabel('Number Of Edges'); ylabel({ 'Change in', 'Mean 44 Segment Depth' }); 45 subplot(2,2,4), hold all, plot(d(3:size(d,1),6), ddd); 46 a = axis; if a(3) < -1, a(3) = -1; end; if a(4) > 1, a(4) = 1; 47 end; axis([30000 45000 a(3) a(4)]); 48 title({'Change in Change in Mean Segment Depth', 'for 49 Biconnected Sub-Graphs of', t1, t2 }); 50 xlabel('Number Of Edges'); ylabel({'Change in Change in', 51 'Mean Segment Depth' }); 52 print(fig, '-dpng', ['DifferentEdges\Biconnected\' upper(53 gn{i}(1)) lower(gn{i}(2:numel(gn{i}))) '- rn{j}(2:numel(rn{j}))) 54 '-MSD'], '-r150'); 55 end 56 end </pre>

Figure	Matlab Code
7.6.1	<pre> 1 plotGraphData(data.edges.ordered.sN4, {'Execution Time of ', '14999 Vertex Ordered Face Trisection Graph (Root = 1/4Nth)'}, 'percentile', 100, 'percentilesteps', 100, 'showbandsinlegend', false, 'showdepthsbestfit', false, 'save', 'DifferentEdges\Biconnected\Ordered-N4-0_43000', 'betweenx', [30000 43000]); </pre>

E.3 Matlab Code To Generate Tables

Table	Matlab Code
6.3.2	<pre>1 n = fieldnames(data.roots); 2 for i=1:numel(n), 3 r = fieldnames(data.roots.(n{i})); 4 for j=1:numel(r), 5 performKS2Tests(data.roots.(n{i}).(r{j})); 6 end 7 end</pre>

E.4 Matlab Functions

E.4.1 appendPercentiles.m

```

1 function [dataout] = appendPercentiles( datain, xcol, ycol, sortOrder )
2 % Appends a column to the data containing grouped percentile values relating to
  % the y-column data grouped on the x-column data
3 %
4 %   Usage:
5 %       dataout = appendPercentiles( datain, xcol, ycol, [A ...] );
6 %
7 %   Will sort the data by the x-column then y-column and will extend the
8 %   matrix by one column that will contain percentiles.
9 %   The resulting matrix will be sorted (if sortOrder is not empty) as
10 %   specified in sortOrder variable (see the sortrows function for more
11 %   details) before returning.
12 sortedData = sortrows( datain, [xcol ycol] );
13 x          = sortedData(:, xcol);
14 numel      = grpstats( x, x, 'numel' );
15 dataout    = [sortedData ones( size(x) )];
16 i          = 1;
17 j          = size( dataout, 2 );
18 for n=1:size( numel ),
19     for m=1:numel(n),
20         dataout(i, j) = 100*m/numel(n);
21         i = i + 1;
22     end;
23 end;
24 if ( ~isempty( sortOrder ) )
25     dataout = sortrows( dataout, sortOrder );
26 end;

```

E.4.2 filldata.m

```

1 function [d] = filldata(x, y)
2 % Fill data
3 ux = unique(x);
4
5 if any( size(ux) ~= size(y) ),
6     disp( size(ux) );
7     disp( size(y) );
8     error('MyToolbox:FILLDATA:IncorrectSize', ...
9         'FILLDATA(X,Y): unique(X) and Y must be the same dimensions')
10 end;
11
12 n      = size(x, 1);
13 sorted = sortrows(x, 1);
14 usorted = sortrows([ux y], 1);
15 d      = ones(size(x));
16 p      = 1;
17 for k=1:n
18     if sorted(k, 1) ~= usorted(p, 1), p = p + 1; end;
19     d(k,1) = usorted(p,2);
20 end;

```

E.4.3 fitdata2.m

```

1 function [vargout] = fitdata2(x, y, whichfits, varargin)
2 %FITDATA2 Fit data to various expressions.
3 %
4 % Usage:
5 %     bestfit = fitdata2(X, Y, WHICHFITS, [ParamName, ParamValue, ...]);
6 %
7 % Where X and Y are Nx1 matrices, respectively, containing the x and y
8 % data upon which a best fit is to be performed. WHICHFITS is a cell
9 % array that can contain character string names of best fits or a nested
10 % cell array containing best fit names. Valid names for best fits
11 % include:
12 %     'ones'      1
13 %     'x'         x
14 %     'x2'        x^2
15 %     'x3'        x^3
16 %     'x4'        x to the power 4
17 %     'xlogx'     x.log(x)
18 %     'xloglogx'  x.log(log(x))
19 %     'xlnx'      x.ln(x)
20 %     'xlnlnx'    x.ln(ln(x))
21 %     'logx'      log(x)
22 %     'loglogx'   log(log(x))
23 %     'lnx'       ln(x)
24 %     'lnlnx'     ln(ln(x))
25 %     'log2x'     log2(x)
26 %     'loglog2x'  log2(log2(x))
27 % Passing a nested cell array of valid names within WHICHFITS will create
28 % an expression formed of all the names and the best fit will be
29 % evaluated against this.
30 %
31 % ParamName and ParamValue are optional pairs of arguments that set
32 % additional options. These can include:
33 %     'useRobustFit'    boolean (default: true)
34 %
35 % The output is a struct with following fields:
36 %     coefficients      A struct containing one field for each best fit;
37 %                       each field contains an array of coefficients for
38 %                       each term of the equation, starting with the
39 %                       constant term and then including all specified
40 %                       terms in order.
41 %     cilower           A struct containing one field for each best fit;
42 %                       each field contains a [Mx1] array where M =
43 %                       size(unique(x),1) containing data points for the
44 %                       lower bounds of the 95% confidence interval for the
45 %                       best fit corresponding to the values in unique(x).
46 %                       THIS IS OT FULLY TESTED AND MAY GIVE INCORRECT
47 %                       BOUNDS.
48 %     ciupper           A struct containing one field for each best fit;
49 %                       each field contains a [Mx1] array where M =
50 %                       size(unique(x),1) containing data points for the
51 %                       upper bounds of the 95% confidence interval for the
52 %                       best fit corresponding to the values in unique(x).
53 %                       THIS IS OT FULLY TESTED AND MAY GIVE INCORRECT
54 %                       BOUNDS.
55 %     data              A struct containing one field for each best fit;
56 %                       each field contains a [Mx1] array where M =
57 %                       size(unique(x),1) containing the best fit data
58 %                       points corresponding to the values in unique(x).
59 %     equation          A struct containing one field for each best fit;
60 %                       each field contains a string representation of the
61 %                       equation of the best fit curve
62 %     mae               A struct containing one field for each best fit;

```

```

63 %           each field contains the Mean Absolute Error (MAE)
64 %           for the best fit.
65 %           rmse       A struct containing one field for each best fit;
66 %           each field contains the Root Mean Squared Error
67 %           (RMSE) for the best fit.
68 %
69 % Example:
70 %     x = (-10:10)';
71 %     y = 2*x.^3 - 38*x + 19;
72 %     y(21) = 0;
73 %     bf = fitdata2( x, y, { 'x3', 'x2', 'x', {'x3','x2'} , ...
74 %           { 'x3', 'x2', 'x' } }, 'userobustfit', true );
75 %     figure; hold all;
76 %     plot( x, y, '+', 'MarkerSize', 3, 'DisplayName', 'Data' );
77 %     fn = fieldnames(bf.data);
78 %     for f=1:numel(fn),
79 %         plot( unique(x), bf.data.(fn{f}), '- ', ...
80 %             'DisplayName', bf.equation.(fn{f}) );
81 %     end;
82 %     legend( 'show', 'Location', 'best' );
83 %     bf.rmse
84 %     bf.mae
85 %
86 % Change 'userobustfit' option to false to see the difference in fit. The
87 % difference between the robustness of the metrics is also apparent as RMSE
88 % shows that the optimal fit is the  $y=ax_3+bx_2+c$  curve whereas MAE correctly
89 % identifies the  $y=ax_3+bx_2+cx+d$  curve as the optimal fit.
90 %
91
92 if nargin < 3,
93     error('MyToolbox:FITDATA2:TooFewInputs',...
94         'FITDATA2 requires three arguments. ');
95 end;
96
97 if any( size(x) ~= size(y) ),
98     error('MyToolbox:FITDATA2',...
99         'FITDATA2 x and y must be the same dimensions')
100 end;
101
102 if isempty(whichfits),
103     fitnames = { 'x', 'xlogx', 'xloglogx' };
104 else
105     fitnames = checkfitname(whichfits, true);
106 end
107
108 options = struct( 'userobustfit', true );
109
110 options = processOptions( options, 'Graphs:PLOTGRAPHDATA', varargin );
111
112 n = size(x, 1);
113 ux = unique(x);
114
115 for k=1:numel(fitnames)
116     names = fitnames{k};
117     if ischar(names),
118         name = names;
119     else
120         name = names{1};
121         if numel(names) > 1,
122             for l=2:numel(names)
123                 name = strcat(name, '_', names{l});
124             end;
125         end;
126     end;
127     d = [ones(size(x)) generatedata(x, fitnames{k})];

```



```

128     ud = [ones(size(ux)) generatedata(ux, fitnames{k})];

130     if options.userobustfit,
131         [vargout.coefficients.(name),s] = robustfit( d(:,2:size(d,2)), y );
132         ci = ...
133             nlparci(vargout.coefficients.(name),s.resid,'cov',s.covb);
134         vargout.rmse.(name) = sqrt(sum(s.resid.^2)/n);
135         vargout.mae.(name) = sum(abs(s.resid))/n;
136         clear s;
137     else
138         [vargout.coefficients.(name), ci, residuals] ...
139             = regress(y, d);
140         vargout.rmse.(name) = sqrt(sum(residuals.^2)/n);
141         vargout.mae.(name) = sum(abs(residuals))/n;
142         clear residuals;
143     end;
144     vargout.cilower.(name) = ud*ci(:,1);
145     vargout.ciuupper.(name) = ud*ci(:,2);
146     clear ci;
147     vargout.data.(name) = ud*vargout.coefficients.(name);
148     coefficients = vargout.coefficients.(name);
149     coef = regexp( name, '-', 'split' );
150     text = [];
151     for c=2:numel(coefficients),
152         if isempty( text ),
153             text = num2str(coefficients(2));
154         else
155             text = strcat( text, formatNumber(coefficients(c)) );
156         end;
157         switch char( coef(c-1) ),
158             case 'x', text = strcat( text, 'x' );
159             case 'x2', text = strcat( text, 'x^2' );
160             case 'x3', text = strcat( text, 'x^3' );
161             case 'x4', text = strcat( text, 'x^4' );
162             case 'lnx', text = strcat( text, 'ln(x)' );
163             case 'lnlnx', text = strcat( text, 'ln(ln(x))' );
164             case 'xlnx', text = strcat( text, 'x{\cdots}ln(x)' );
165             case 'xlnlnx', text = strcat( text, 'x{\cdots}ln(ln(x))' );
166             case 'logx', text = strcat( text, 'log(x)' );
167             case 'loglogx', text = strcat( text, 'log(log(x))' );
168             case 'log2x', text = strcat( text, 'log_2(x)' );
169             case 'loglog2x', text = strcat( text, 'log_2(log_2(x))' );
170             case 'xlogx', text = strcat( text, 'x{\cdots}log(x)' );
171             case 'xloglogx', text = strcat( text, 'x{\cdots}log(log(x))' );
172         end;
173     end;
174     vargout.equation.(name) = strcat( 'y=', text, formatNumber(coefficients(1)) );
175 end

177 function [data] = generatedata(x, name)
178 %GENERATEDATA Generates fit data.
179 if ischar(name),
180     switch name
181         case 'ones', data = [];
182         case 'x', data = x;
183         case 'x2', data = x.^2;
184         case 'x3', data = x.^3;
185         case 'x4', data = x.^4;
186         case 'lnx', data = log(x);
187         case 'lnlnx', data = log(log(x));
188         case 'xlnx', data = x.*log(x);
189         case 'xlnlnx', data = x.*log(log(x));
190         case 'logx', data = log10(x);
191         case 'loglogx', data = log10(log10(x));

```

```

192         case 'log2x',      data = log2(x);
193         case 'loglog2x',   data = log2(log2(x));
194         case 'xlogx',      data = x.*log10(x);
195         case 'xloglogx',   data = x.*log10(log10(x));
196         otherwise,        data = [];
197     end
198 else
199     data = [];
200     for j=1:numel(name)
201         data = [data generatedata(x, name{j})];
202     end
203 end;
204 end

206 function [fn] = checkfitname(name, propagate)
207 %CHECKFITNAME Checks to see if the name of the fit is valid.
208 if nargin == 1,
209     propagate = true;
210 end

212 if iscellstr(name),
213     fn = {};
214     for j=1:numel(name)
215         fit = checkfitname(name{j}, false);
216         if ~isempty(fit),
217             fn{numel(fn) + 1} = fit;
218         end;
219     end;
220 elseif iscell(name),
221     if propagate,
222         fn = {};
223         for j=1:numel(name)
224             fit = checkfitname(name{j}, false);
225             if ~isempty(fit),
226                 fn{numel(fn) + 1} = fit;
227             end;
228         end;
229     else
230         fn = [];
231     end;
232 elseif ischar(name),
233     switch name
234     case { 'ones', 'x', 'x2', 'x3', 'x4', 'xlnx', 'xlnlnx', 'lnx',
235           'lnlnx', 'xlogx', 'xloglogx', 'logx', 'loglogx', 'log2x', 'loglog2x'
236         }
237         fn = name;
238     otherwise
239         warning('MyToolbox:FITDATA2','Ignoring unknown fit: %s',name);
240         fn = [];
241     end;
242 else
243     fn = [];
244 end;
245 end

247 function [str] = formatNumber( n )
248 if n < 0,
249     str = num2str(n);
250 else
251     str = ['+' num2str(n)];
252 end;
253 end
254 end

```

E.4.4 fitdatatoarray.m

```

1 function [vargout] = fitdatatoarray(x, y, ya, varargin)
2 % Fit data to an array.
3 %
4 % Usage:
5 %     bestfit = fitdatatoarray( X, Y, YA, [ParamName, ParamValue, ...] );
6 %
7 % Where X and Y are Nx1 matrices and YA is a Mx1 matrix where M is the
8 % number of unique elements in X.
9 %
10 % ParamName and ParamValue are optional pairs of arguments that set
11 % additional options. These can include:
12 %     'useRobustFit'      boolean (default: true)
13 %
14 % The output is a struct with following fields:
15 %     coefficients      A struct containing one field for each best fit;
16 %                       each field contains an array of coefficients for
17 %                       each term of the equation, starting with the
18 %                       constant term and then including all specified
19 %                       terms in order.
20 %     cilower            A struct containing one field for each best fit;
21 %                       each field contains a [Mx1] array where M =
22 %                       size(unique(x),1) containing data points for the
23 %                       lower bounds of the 95% confidence interval for the
24 %                       best fit corresponding to the values in unique(x).
25 %                       THIS IS OT FULLY TESTED AND MAY GIVE INCORRECT
26 %                       BOUNDS.
27 %     ciupper           A struct containing one field for each best fit;
28 %                       each field contains a [Mx1] array where M =
29 %                       size(unique(x),1) containing data points for the
30 %                       upper bounds of the 95% confidence interval for the
31 %                       best fit corresponding to the values in unique(x).
32 %                       THIS IS OT FULLY TESTED AND MAY GIVE INCORRECT
33 %                       BOUNDS.
34 %     data               A struct containing one field for each best fit;
35 %                       each field contains a [Mx1] array where M =
36 %                       size(unique(x),1) containing the best fit data
37 %                       points corresponding to the values in unique(x).
38 %     equation           A struct containing one field for each best fit;
39 %                       each field contains a string representation of the
40 %                       equation of the best fit curve
41 %     mae                A struct containing one field for each best fit;
42 %                       each field contains the Mean Absolute Error (MAE)
43 %                       for the best fit.
44 %     rmse               A struct containing one field for each best fit;
45 %                       each field contains the Root Mean Squared Error
46 %                       (RMSE) for the best fit.
47 %
48 % Example:
49 %     x1 = (-10:10)';
50 %     x  = [x1; x1; x1];
51 %     y  = 0.01*x.^3-x + 23 + 0.4*randn(63,1);
52 %     ya = sin(-pi:pi/10:pi)';
53 %     bf = fitdatatoarray( x, y, ya, 'userobustfit', true );
54 %     figure; hold;
55 %     plot( x, y, 'b+', 'DisplayName', 'Data', 'MarkerSize', 3 );
56 %     plot( unique(x), bf.data, 'r-', 'DisplayName', bf.equation);
57 %     legend( 'Show', 'Location', 'Best' );
58 %     bf.rmse
59 %     bf.mae
60 %
61
62 if any( size(x) ~= size(y) ),

```

```

63     error('MyToolbox:FITDATATOARRAY',...
64           'FITDATATOARRAY x and y must be the same dimensions')
65 end;

67 ux = unique(x);

69 if any( size(ux)~=size(ya) ),
70     error('MyToolbox:FITDATATOARRAY:IncorrectSize',...
71           'FITDATATOARRAY(X,Y,YA): unique(X) and YA must be the same dimensions')
72 end;

74 options = struct( 'userobustfit', true, ...
75                   'terms',          {} );

77 options = processOptions( options, 'Graphs:PLOTGRAPHDATA', varargin );

79 terms = options.terms;
80 for k=1:numel(terms),
81     switch lower( terms{k} ),
82         case { 'x', 'lnx', 'lnlnx', 'x2', 'x3', 'x4', 'xlnx', 'xlnlnx' },
83             otherwise, terms(k) = [];
84     end;
85 end;

87 n      = size(x, 1);
88 sorted = sortrows([x ya], 1);
89 unsorted = sortrows([ux ya], 1);
90 d        = [ones(size(x)) zeros( size(x, 1), 2*numel(terms) + 1 )];
91 ud       = [ones(size(ux)) ya zeros( size( ux, 1 ), 2*numel(terms) )];
92 p        = 1;
93 for k=1:n
94     if sorted(k, 1)~=unsorted(p, 1), p = p + 1; end;
95     d(k,2) = unsorted(p,2);
96 end;

98 for k=1:numel(terms),
99     switch lower( terms{k} ),
100         case 'x',
101             d(:, 1+((k*2):(k*2+1))) = d(:, 1:2).*[x x];
102             ud(:, 1+((k*2):(k*2+1))) = ud(:, 1:2).*[ux ux];
103         case 'lnx',
104             d(:, 1+((k*2):(k*2+1))) = d(:, 1:2).*[log(x) log(x)];
105             ud(:, 1+((k*2):(k*2+1))) = ud(:, 1:2).*[log(ux) log(ux)];
106         case 'lnlnx',
107             d(:, 1+((k*2):(k*2+1))) = d(:, 1:2).*[log(log(x))];
108             ud(:, 1+((k*2):(k*2+1))) = ud(:, 1:2).*[log(log(ux)) log(log(ux))];
109         case 'x2',
110             d(:, 1+((k*2):(k*2+1))) = d(:, 1:2).*(x.^2);
111             ud(:, 1+((k*2):(k*2+1))) = ud(:, 1:2).*[ux.^2 ux.^2];
112         case 'x3',
113             d(:, 1+((k*2):(k*2+1))) = d(:, 1:2).*(x.^3);
114             ud(:, 1+((k*2):(k*2+1))) = ud(:, 1:2).*[ux.^3 ux.^3];
115         case 'x4',
116             d(:, 1+((k*2):(k*2+1))) = d(:, 1:2).*(x.^4);
117             ud(:, 1+((k*2):(k*2+1))) = ud(:, 1:2).*[ux.^4 ux.^4];
118         case 'xlnx',
119             d(:, 1+((k*2):(k*2+1))) = d(:, 1:2).*(x.*log(x));
120             ud(:, 1+((k*2):(k*2+1))) = ud(:, 1:2).*[ux.*log(ux) ux.*log(ux)];
121         case 'xlnlnx',
122             d(:, 1+((k*2):(k*2+1))) = d(:, 1:2).*(x.*log(log(x)));
123             ud(:, 1+((k*2):(k*2+1))) = ud(:, 1:2).*[ux.*log(log(ux))
124                                     ux.*log(log(ux))];
125     end;
126 end;
127 if options.userobustfit ,

```

```

127     [vargout.coefficients,s] = robustfit( d(:,2:size(d,2)), sorted(:,2) );
128     ci                        = nlparci(vargout.coefficients, s.resid, ...
129         'cov', s.covb);
130     vargout.isRobustFit      = true;
131     vargout.rmse            = sqrt(sum(s.resid.^2)/n);
132     vargout.mae             = sum(abs(s.resid))/n;
133     clear s;
134 else
135     [vargout.coefficients, ci, residuals] = regress(sorted(:,2), d);
136     vargout.isRobustFit      = false;
137     vargout.rmse            = sqrt(sum(residuals.^2)/n);
138     vargout.mae             = sum(abs(residuals))/n;
139     clear residuals;
140 end;

142 vargout.cilower            = ud*ci(:,1);
143 vargout.ciupper           = ud*ci(:,2);
144 clear ci d;
145 vargout.data               = ud*vargout.coefficients;
146 vargout.equation           = ['y=' num2str(vargout.coefficients(2)) 'f(x)' ...
147     formatNumber(vargout.coefficients(1))];

149 for k=1:numel(terms),
150     switch lower( terms{k} ),
151         case 'x',          eqn = 'x';
152         case 'lnx',        eqn = 'ln(x)';
153         case 'lnlnx',      eqn = 'ln(ln(x))';
154         case 'x2',         eqn = 'x^2';
155         case 'x3',         eqn = 'x^3';
156         case 'x4',         eqn = 'x^4';
157         case 'xlnx',       eqn = 'x{\cdots}ln(x)';
158         case 'xlnlnx',     eqn = 'x{\cdots}ln(ln(x))';
159     end;
160     vargout.equation = strcat( vargout.equation, ['+' eqn '{\cdots}(' ...
161         formatNumber(vargout.coefficients(2*k+2)) 'f(x)' ...
162         formatNumber(vargout.coefficients(2*k+1)) ')'] );
163 end;

165 function [str] = formatNumber( n )
166     if n < 0,
167         str = num2str(n);
168     else
169         str = ['+' num2str(n)];
170     end;
171 end
172 end

```

E.4.5 loadAndSplitGraphData.m

```

1 function [data] = loadAndSplitGraphData( data, path, inputID, type, ...
2     splitOn, splitIDs, outputIDs )
3 %LOADANDSPLITGRAPHDATA Loads Graph Data and separates out errors and
4 %garbage collections.
5 %
6 % Usage:
7 %     [data] = loadGraphData( P, ID, T );
8 %
9 % Where: P is a character array in the format: '<File Directory>\<File
10 % Stub>'; ID is a cell array containing character strings; and T is one
11 % of 'vertex', 'edge', 'root'.
12 %
13 % Example:
14 %     [data] = loadGraphData( 'Data\SameRootsGraph-', { '50', '100' },
15 %         'root' );
16 %
17 % Will try to load and process the files:
18 %     'Data\SameRootsGraph-50_Total.log';
19 %     'Data\SameRootsGraph-50_Depths.log';
20 %     'Data\SameRootsGraph-100_Total.log'; and
21 %     'Data\SameRootsGraph-100_Depths.log'
22 %
23 % The Totals log file should have the following columns:
24 %     1) Number of Vertices;
25 %     2) Number of Edges;
26 %     3) Root Vertex ID;
27 %     4) Execution Time (ns); and
28 %     5) Number of Garbage Collections;
29 %
30 % The following columns are added to the data:
31 %     6) Sequentially incremented row index; and
32 %     7) Grouped Percentile for data (grouped using the 'type' column).
33 %
34 % The Depths log file should have the following columns:
35 %     1) Number of vertices/edges or root vertex id for the given 'type'
36 %        argument.
37 %     2) The average segment depth of the graph.
38 %
39 % The output is a struct containing one child element for each element in
40 % the ID cell array (in the format ['s' ID{k}]). Each child has the
41 % following children:
42 %         type           Set to the type argument.
43 %         data.error      Contains rows from the total log file where the
44 %                        execution time is negative.
45 %         data.gc         Contains rows from the total log file where the
46 %                        number of garbage collections is non-zero and
47 %                        the execution time is positive.
48 %         data.gc         Contains rows from the total log file where the
49 %                        number of garbage collections is zero and the
50 %                        execution time is positive.
51 %         data.depths     Contains the rows from the depths log file.
52 %
53 %
54 switch lower( type ),
55     case 'vertex', xcol = 1;
56     case 'edge',   xcol = 2;
57     case 'root',   xcol = 3;
58     otherwise,     xcol = 1;
59                     type = 'vertex';
60 end;
61
62 switch lower( splitOn ),

```

```

63     case 'vertex',    scol = 1;
64     case 'edge',      scol = 2;
65     case 'root',      scol = 3;
66     otherwise,        scol = 1;
67 end;

69 n = load( [path inputID '_Total.log'] );
70 for i=1:numel(splitIDs),
71     depthFile = [path outputIDs{i} '_Depths.log'];
72     name = ['s' outputIDs{i}];
73     data.(name).type = type;
74     [o n] = removeoutliers( n, scol, 4, 'notequaltox', ...
75                             splitIDs{i});
76     [correct data.(name).data.error] ...
77         = removeoutliers( [o (1:size(o,1))'], ...
78                             xcol, 4, 'lessthan', 0 );
79     clear o;
80     [data.(name).data.nogc data.(name).data.gc] = removeoutliers( ...
81                                     appendPercentiles( correct, xcol, 4, 6 ), ...
82                                     xcol, 5, 'greaterthan', 0 );
83     clear correct;
84     data.(name).data.depths = load( depthFile );
85 end;

```

E.4.6 loadGraphData.m

```

1 function [data] = loadGraphData( data, path, ids, type, varargin )
2 %LOADGRAPHDATA Loads Graph Data and separates out errors and garbage
3 %collections.
4 %
5 % Usage:
6 %     [data] = loadGraphData( P, ID, T );
7 %
8 % Where: P is a character array in the format: '<File Directory>\<File
9 % Stub>'; ID is a cell array containing character strings; and T is one
10 % of 'vertex', 'edge', 'root'.
11 %
12 % Example:
13 %     [data] = loadGraphData( 'Data\SameRootsGraph-', { '50', '100' },
14 %                             'root' );
15 %
16 % Will try to load and process the files:
17 %     'Data\SameRootsGraph-50_Total.log';
18 %     'Data\SameRootsGraph-50_Depths.log';
19 %     'Data\SameRootsGraph-100_Total.log'; and
20 %     'Data\SameRootsGraph-100_Depths.log'
21 %
22 % The Totals log file should have the following columns:
23 %     1) Number of Vertices;
24 %     2) Number of Edges;
25 %     3) Root Vertex ID;
26 %     4) Execution Time (ns); and
27 %     5) Number of Garbage Collections;
28 %
29 % The following columns are added to the data:
30 %     6) Sequentially incremented row index; and
31 %     7) Grouped Percentile for data (grouped using the 'type' column).
32 %
33 % The Depths log file should have the following columns:
34 %     1) Number of vertices/edges or root vertex id for the given 'type'
35 %        argument.
36 %     2) The average segment depth of the graph.

```

```

37 %
38 % The output is a struct containing one child element for each element in
39 % the ID cell array (in the format ['s' ID{k}]). Each child has the
40 % following children:
41 %         type          Set to the type argument.
42 %         data.error     Contains rows from the total log file where the
43 %                         execution time is negative.
44 %         data.gc        Contains rows from the total log file where the
45 %                         number of garbage collections is non-zero and
46 %                         the execution time is positive.
47 %         data.gc        Contains rows from the total log file where the
48 %                         number of garbage collections is zero and the
49 %                         execution time is positive.
50 %         data.depths    Contains the rows from the depths log file.
51 %
52 options = struct( 'filepostfix', 'Total' );
53
54 options = processOptions( options, 'Graphs:PLOTGRAPHDATA', varargin );
55
56 switch lower( type ),
57     case 'vertex', xcol = 1;
58     case 'edge',   xcol = 2;
59     case 'root',   xcol = 3;
60     otherwise,     xcol = 1;
61                     type = 'vertex';
62 end;
63
64 for i=1:numel(ids),
65     totalFile = [path ids{i} '_' options.filepostfix '.log'];
66     depthFile = [path ids{i} '_Depths.log'];
67     if ( ~exist( totalFile, 'file' ) )
68         disp( [totalFile ' does not exist.'] );
69         continue;
70     end;
71     if ( ~exist( depthFile, 'file' ) )
72         disp( [depthFile ' does not exist.'] );
73         continue;
74     end;
75     name = ['s' ids{i}];
76     data.(name).type = type;
77     o = load( totalFile );
78     [correct data.(name).data.error] ...
79         = removeoutliers( [o (1:size(o,1))'], ...
80                             xcol, 4, 'lessthan', 0 );
81     clear o;
82     [data.(name).data.nogc data.(name).data.gc] = removeoutliers( ...
83         appendPercentiles( correct, xcol, 4, 6 ), ...
84         xcol, 5, 'greaterthan', 0 );
85     clear correct;
86     data.(name).data.depths = load( depthFile );
87 end;

```


E.4.7 performKS2Tests.m

```

1 function [s] = performKS2Tests( data, varargin )
2
3 options = struct( 'percentile', 100, ...
4                  'mapx', false, ...
5                  'betweenx', [-1 -1], ...
6                  'xlimit', -1, ...
7                  );
8
9 options = processOptions( options, 'Graphs:PLOTGRAPHDATA', varargin );
10
11 switch (data.type)
12     case 'vertex', xcol = 1;
13     case 'edge', xcol = 2;
14     case 'root', xcol = 3;
15     otherwise, warning( 'Graphs:PLOTNUMBEROFELEMENTS', ...
16                        'Unknown Data Type: Assuming vertex' );
17                        xcol = 1;
18 end;
19 s.xcol = xcol;
20
21 d = [data.data.nogc(:, xcol) data.data.nogc(:, 4) data.data.nogc(:, 7)];
22 g = [data.data.gc(:, xcol) data.data.gc(:, 4) data.data.gc(:, 7)];
23
24 if options.percentile <= 0,
25     return;
26 end;
27
28 s.count.nogc.before = size(d,1);
29 s.count.gc.before = size(g,1);
30 if options.percentile < 100,
31     d = removeoutliers( d, 2, 3, 'greaterthany', options.percentile );
32     g = removeoutliers( g, 2, 3, 'greaterthany', options.percentile );
33     s.percentile = options.percentile;
34 else
35     s.percentile = 100;
36 end;
37 s.count.nogc.after = size(d,1);
38 s.count.gc.after = size(g,1);
39
40 if any(options.betweenx ~= [-1 -1]),
41     d = removeoutliers( d, 1, 2, 'notbetweenx', options.betweenx );
42     g = removeoutliers( g, 1, 2, 'notbetweenx', options.betweenx );
43 end;
44
45 if options.mapx && xcol == 3,
46     fnct = inline( 'round(mod(x,3)*y/3)+ceil(x/3)' );
47     d = sortrows( [fnct( d(:,1), max(d(:,1))) ] d(:, 2) zeros(size(d,1),1);
48                 ...
49                 fnct( g(:,1), max(g(:,1))) ] g(:, 2)
50                 ones(size(g,1),1) ] );
51 else
52     d = sortrows( [d(:,1) d(:,2) zeros(size(d,1),1); g(:,1) g(:,2)
53                 ones(size(g,1),1) ] );
54 end;
55
56 x = d(:, 1);
57 y = d(:, 2);
58 ux = unique( x );
59
60 n = grpstats( y, x, 'numel' );
61
62 if options.xlimit > 0,

```

```

60     maxi = min( options.xlimit , numel(n) );
61 else
62     maxi = numel(n);
63 end;

65 s.x      = ux( 1:maxi, 1 );
66 s.nogc   = cell( maxi, 1 );
67 s.gc     = cell( maxi, 1 );

69 p = 0;
70 for i=1:maxi,
71     dd      = d( p + (1:n(i))', : );
72     [gc nogc] = removeoutliers( dd , 1, 3, 'equaltoy', 1 );
73     if numel(nogc) > 0,
74         s.nogc{i} = nogc(:, 2);
75         s.mean.nogc(i) = mean(s.nogc{i});
76         s.sd.nogc(i) = std(s.nogc{i});
77     end;
78     if numel(gc) > 0,
79         s.gc{i} = gc(:, 2);
80         s.mean.gc(i) = mean(s.gc{i});
81         s.sd.gc(i) = std(s.gc{i});
82         if numel(nogc) > 0,
83             s.ks(i) = kstest2( s.nogc{i}, s.gc{i} );
84             s.cm(i) = cmtest2( s.nogc{i}, s.gc{i} );
85             % s.k(i) = kuipertest2( s.nogc{i}, s.gc{i} );
86             % full = [s.nogc{i};s.gc{i}];
87             % groups = [ones(size(s.nogc{i})); zeros(size(s.gc{i}))];
88             % s.kw(i) = kruskalwallis( full , groups , 'off' );
89             % s.an(i) = anova1( full , groups , 'off' );
90         end;
91     end;
92     p = p + n(i);
93 end;
94 if isfield( s , 'ks' ),
95     fprintf( '\t&\n' );
96     fprintf( '\t%d &\n' , numel(s.x) );
97     fprintf( '\t%d (%3.3f%%) &\n' , sum(s.ks) , (100*sum(s.ks)/numel(s.ks)) );
98     fprintf( '\t%d (%3.3f%%)\n' , sum(s.cm) , (100*sum(s.cm)/numel(s.cm)) );
99     % fprintf( '\t%d (%3.3f%%)\n' , sum(s.k) , (100*sum(s.k)/numel(s.k)) );
100    fprintf( '\\\tabularnewline\n' );
101 else
102    disp( [ 'No data: No GC = ' num2str(s.count.nogc.after) '/' ...
103          num2str(s.count.nogc.before) ' (' ...
104          num2str(100*s.count.nogc.after/s.count.nogc.before) ...
105          '%), GC = ' num2str(s.count.gc.after) '/' ...
106          num2str(s.count.gc.before) ' (' ...
107          num2str(100*s.count.gc.after/s.count.gc.before) '%)' ] );
108 end;

```

E.4.8 plotGraphData.m

```

1 function plotGraphData( data, titleText, varargin )
2
3 options = struct( 'type', 'nogc', ...
4                  'percentile', 90, ...
5                  'percentilesteps', 30, ...
6                  'lowcolour', [0 0 0], ...
7                  'highcolour', [.7 .7 .7], ...
8                  'newfigure', true, ...
9                  'mapx', false, ...
10                 'plotsegments', false, ...
11                 'usehsl', false, ...
12                 'flipdepths', false, ...
13                 'showbestfits', true, ...
14                 'showdepthsbestfit', true, ...
15                 'showsegmentsbestfit', false, ...
16                 'showdepthsumbestfit', false, ...
17                 'showbestfitrmse', false, ...
18                 'showbestfitmae', true, ...
19                 'showbestfitcis', false, ...
20                 'bestfits', {{ 'x', 'xlnx', 'xlnlnx' }}, ...
21                 'userobustfit', true, ...
22                 'showbandsinlegend', true, ...
23                 'save', '', ...
24                 'betweenx', [-1 -1], ...
25                 'thresholdlimit', [0 0], ...
26                 'showthresholdlimit', false, ...
27                 'setthresholdlimit', false, ...
28                 'scalethreshold', 0 );
29
30 options = processOptions( options, 'Graphs:PLOTGRAPHDATA', varargin );
31
32 options.percentile = min( max( options.percentile, 0 ), 100 );
33
34 if options.percentile == 0,
35     warning( 'Graphs:PLOTGRAPHDATA', ...
36             'Percentile is zero; no data to plot.' )
37     return;
38 end;
39
40 switch lower( options.type ),
41     case 'nogc', d = data.data.nogc;
42     case 'gc', d = data.data.gc;
43     case 'full', d = [data.data.nogc; data.data.gc];
44     otherwise, warning( 'Graphs:PLOTGRAPHDATA', ...
45                         'Invalid type, assuming "nogc".' );
46                 d = data.data.nogc;
47 end;
48
49 if options.newfigure,
50     screenSize = get(0,'ScreenSize');
51 % figure( 'Position', [screenSize(3)/6 screenSize(4)/6 2*screenSize(3)/3
52 % 2*screenSize(4)/3], 'PaperSize', [20.98 29.68], 'Color', 'white');
53 fig = figure( 'Position', [0 0 screenSize(3) screenSize(4)], 'PaperUnits',
54               'inches', 'PaperPosition', [0 0 10 7], 'Color', 'white');
55 else
56     fig = gcf;
57 end;
58
59 switch (data.type)
60     case 'vertex', xcol = 1;
61     case 'edge', xcol = 2;
62     case 'root', xcol = 3;

```

```

61     otherwise,      warning( 'Graphs:PLOTGRAPHDATA', 'Unknown Data Type:
        Assuming vertex' );
62                     xcol = 1;
63 end;

65 depths = data.data.depths;

67 if size(depths,2) < 3,
68     if options.plotsegments,
69         disp( 'plotsegments = false' );
70         options.plotsegments = false;
71     end
72     if options.showsegmentsbestfit,
73         disp( 'showsegmentsbestfit = false' );
74         options.showsegmentsbestfit = false;
75     end
76     if options.showdepthsumbestfit,
77         disp( 'showdepthsumbestfit = false' );
78         options.showdepthsumbestfit = false;
79     end
80 end

82 if options.plotsegments,
83     d = [d filldata(d(:,xcol), depths(:,3))];
84     xcol = size(d,2);
85 end

87 if options.mapx && xcol == 3,
88     fnct = inline( 'ceil((mod(x,3)*y+x)/3)' );
89 else
90     fnct = [];
91 end;

93 if ~isempty( titleText ),
94     if options.percentile < 100,
95         t = [num2str(options.percentile) 'th Percentile'];
96     else
97         t = '';
98     end;
99     if strcmp( t, '' ),
100         title( titleText );
101     else
102         title( [titleText ' ( ' t ')'] );
103     end;
104     switch (xcol)
105     case 1, xlabel( 'Number of Vertices' );
106     case 2, xlabel( 'Number of Edges' );
107     case 3, if options.mapx,
108 %             xlabel( ['$\displaystyle\left\lceil\frac{\right.
num2str(max(d(:,xcol))) '\rm modulo}(\{\rm Root Vertex ID\},3) + \{\rm Root Vertex
ID\}\right\rceil$'], 'interpreter', 'latex' );
109                 xlabel( 'Root Vertex ID (Mapped)' );
110             else
111                 xlabel( 'Root Vertex ID' );
112             end;
113     case size(d,2), xlabel( 'Number of Segments' );
114     end;
115     ylabel( 'Execution Time (ns)' );
116 end;

118 if xcol == 2,
119     depths = [unique(d(:, xcol)) depths];
120 end

122 if options.percentile < 100,

```

```

123     d = removeoutliers( d, xcol, 7, 'greaterthany', options.percentile );
124 end;

126 if options.usethresholdlimit && any( options.thresholdlimit ~= 0 ),
127     [d o] = removeoutliers( d, xcol, 4, 'greaterthanlinear',
128         options.thresholdlimit );
129 else
129     o = [];
130 end;

132 if any(options.betweenx ~= [-1 -1]),
133     d = removeoutliers( d, xcol, 4, 'notbetweenx', options.betweenx );
134 end;

136 if xcol ~= 2,
137     depths = removeoutliers( depths, 1, 2, 'notismemberx', unique( d(:, 3) ) );
138 else
139     depths = removeoutliers( depths, 1, 2, 'notismemberx', unique( d(:, 2) ) );
140     depths = depths(:, 2:size(depths,2));
141 end

143 if options.percentilesteps == 0,
144     options.percentilesteps = options.percentile / 16;
145 end;

147 colourSteps      = max(ceil( options.percentile / options.percentilesteps ), 1);
148 colourMatrix      = getColoursBetween( options.lowcolour, options.highcolour,
149     colourSteps, options.usehsl );

150 hold on;
151 e = d;
152 prevBounds = options.percentile;
153 for j=colourSteps:-1:1,
154     if j > 1,
155         bounds = options.percentilesteps * (j - 1);
156         [f e] = removeoutliers( e, xcol, 7, 'lessthany', bounds );
157     else
158         bounds = 0;
159         f = e;
160     end;
161     x = f(:, xcol);
162     if options.mapx,
163         x = fnct(x,max(x));
164     end;
165     if options.showbandsinlegend,
166         displayName = [ num2str(bounds) '-' num2str(prevBounds) '^{th} %ile
167             Data'];
168         prevBounds = bounds;
169     else
170         displayName = '';
171     end;
172     handle = plot( x, f(:,4), '.*', 'Color', colourMatrix(j,:), 'MarkerSize', 1,
173         'DisplayName', displayName);
174     if ~options.showbandsinlegend,
175         hasbehavior( handle, 'legend', false );
176     end;
177 end;

177 if ~isempty(o) && options.scalethreshold ~= 0,
178     plot( o(:, xcol), o(:, 4)*options.scalethreshold, '.*', 'MarkerSize', 1,
179         'Color', [1 0 0], 'DisplayName', ['Outlying Data (Scale Factor = ' num2str(
180     options.scalethreshold ) ']' );
179 end;

181 fiterrors = '';

```

```

183 hold all;
184 if options.showbestfits || options.showdepthsbestfit ||
options.showsegmentsbestfit || options.showdepthsumbestfit ,
185     x = d(:,xcol);
186     y = d(:,4);
187     switch (xcol)
188     case {1, 2, size(d,2)}, % Vertex or Edge
189         bf = fitdata2( x, y, options.bestfits, 'userobustfit',
options.userobustfit );
190         ux = unique( x );
191         fn = fieldnames( bf.rmse );
192         num_fn = numel(fn);
193         nc = 0;
194         if options.showbestfits ,
195             nc = nc + num_fn;
196         end;
197         if options.showdepthsbestfit ,
198             nc = nc + 1;
199         end;
200         if options.showdepthsumbestfit ,
201             nc = nc + 1;
202         end;
203         if options.showsegmentsbestfit ,
204             nc = nc + num_fn;
205         end;
206         if options.showthresholdlimit ,
207             nc = nc + 1;
208         end;
209         Colours = hsl2rgb([(0:1/nc:1-1/nc)' ones(nc,1) ones(nc,1)/2]);
210         colourNum = 0;
211         if options.showbestfits ,
212             for b=1:num_fn,
213                 colourNum = colourNum + 1;
214                 plot( ux, bf.data.(fn{b}), '-', 'DisplayName',
getLabelText( bf, fn{b}, options.userobustfit ,
options.showbestfitrmse , options.showbestfitmae ),
'Color', Colours(colourNum,:) );
215                 if options.showbestfitrmse ,
216                     fiterrors = [fiterrors ' ' num2str( round(
bf.rmse.(fn{b}) ) ) ' &' ]; %ok<AGROW>
217                 end;
218                 if options.showbestfitmae ,
219                     fiterrors = [fiterrors ' ' num2str( round(
bf.mae.(fn{b}) ) ) ' &' ]; %ok<AGROW>
220                 end;
221                 if options.showbestfitcis ,
222                     handle = plot( ux, bf.cilower.(fn{b}), '--',
'Color', Colours(colourNum,:) );
223                     hasbehavior( handle, 'legend', false );
224                     handle = plot( ux, bf.ciupper.(fn{b}), '--',
'Color', Colours(colourNum,:) );
225                     hasbehavior( handle, 'legend', false );
226                 end;
227             end;
228         end;
229         if options.showdepthsbestfit ,
230             colourNum = colourNum + 1;
231             depth = depths(:,2);
232             if options.flipdepths ,
233                 depth = flipud( depth );
234             end
235             dbf = fitdatatoarray( x, y, depth, 'userobustfit',
options.userobustfit );

```

```

236         plot( ux, dbf.data, '-', 'DisplayName', getLabelText( dbf,
237             'depths', options.userobustfit, options.showbestfitrmse,
238             options.showbestfitmae ), 'Color', Colours(colourNum, :) );
239         if options.showbestfitrmse,
240             fiterrors = [fiterrors ' ' num2str( round( dbf.rmse ) ) '
241                 &' ];
242         end;
243         if options.showbestfitmae,
244             fiterrors = [fiterrors ' ' num2str( round( dbf.mae ) ) '
245                 &' ];
246         end;
247     end;
248     if options.showdepthsumbestfit,
249         colourNum = colourNum + 1;
250         depth = depths(:,2) .* depths(:,3);
251         if options.flipdepths,
252             depth = flipud( depth );
253         end
254         dbf = fitdatatoarray( x, y, depth, 'userobustfit',
255             options.userobustfit );
256         plot( ux, dbf.data, '-', 'DisplayName', getLabelText( dbf,
257             'depthsum', options.userobustfit, options.showbestfitrmse,
258             options.showbestfitmae ), 'Color', Colours(colourNum, :) );
259         if options.showbestfitrmse,
260             fiterrors = [fiterrors ' ' num2str( round( dbf.rmse ) ) '
261                 &' ];
262         end;
263         if options.showbestfitmae,
264             fiterrors = [fiterrors ' ' num2str( round( dbf.mae ) ) '
265                 &' ];
266         end;
267     end;
268     if options.showsegmentsbestfit,
269         s = filldata(x, depths(:,3));
270         sf = fitdata2( s, y, options.bestfits, 'userobustfit',
271             options.userobustfit );
272         % sf = fitdata3( x, depths(:,3), y, options.bestfits,
273         % 'userobustfit', options.userobustfit );
274         for b=1:num_fn,
275             plot( ux, sf.data.(fn{b}), '-', 'DisplayName',
276                 regexp( getLabelText( sf, fn{b},
277                     options.userobustfit, options.showbestfitrmse,
278                     options.showbestfitmae ), 'x', 's'), 'Color',
279                     Colours(colourNum + b, :) );
280             if options.showbestfitrmse,
281                 fiterrors = [fiterrors ' ' num2str( round(
282                     sf.rmse.(fn{b}) ) ) ' &' ]; %ok<AGROW>
283             end;
284             if options.showbestfitmae,
285                 fiterrors = [fiterrors ' ' num2str( round(
286                     sf.mae.(fn{b}) ) ) ' &' ]; %ok<AGROW>
287             end;
288         end
289     end;
290     if options.showthresholdlimit,
291         a = axis;
292         plot( [a(1) a(2)], [a(1) a(2)].*options.thresholdlimit(1) +
293             options.thresholdlimit(2), '-', 'DisplayName', 'Threshold
294             Limit', 'Color', Colours(nc, :) );
295     end;
296 case 3, % Root
297     cf = fitdata2( x, y, { 'ones' }, 'userobustfit',
298         options.userobustfit );

```

```

279         plot( unique(x), cf.data.ones, '-', 'DisplayName', getLabelText(
        cf, 'ones', options.userobustfit, options.showbestfitmse,
        options.showbestfitmae ) );
280     if options.showbestfitmse,
281         fiterrors = [fiterrors ' ' num2str( round( cf.rmse.ones ) )
        ' &' ]; %#ok<AGROW>
282     end;
283     if options.showbestfitmae,
284         fiterrors = [fiterrors ' ' num2str( round( cf.mae.ones ) ) '
        &' ]; %#ok<AGROW>
285     end;
286     if options.showdepthsbestfit,
287         bf = fitdatatoarray( x, y, depths(:,2), 'userobustfit',
        options.userobustfit );
288         ux = unique( x );
289         if ( ~isempty( fnct ) )
290             fit = sortrows([fnct(ux,max(ux)) bf.data], 1 );
291             plot( fit(:,1), fit(:,2), '-', 'DisplayName',
        getLabelText( bf, 'depths', options.userobustfit,
        options.showbestfitmse, options.showbestfitmae ) );
292         else
293             plot( ux, bf.data, '-', 'DisplayName', getLabelText( bf,
        'depths', options.userobustfit, options.showbestfitmse,
        options.showbestfitmae ) );
294         end;
295         if options.showbestfitmse,
296             fiterrors = [fiterrors ' ' num2str( round( bf.rmse ) ) '
        &' ]; %#ok<AGROW>
297         end;
298         if options.showbestfitmae,
299             fiterrors = [fiterrors ' ' num2str( round( bf.mae ) ) '
        &' ]; %#ok<AGROW>
300         end;
301     end
302     if options.showthresholdlimit,
303         a = axis;
304         plot( [a(1) a(2)], [a(1) a(2)].*options.thresholdlimit(1) +
        options.thresholdlimit(2), '-', 'DisplayName', 'Threshold
        Limit' );
305     end;
306 end;
307 else
308     if options.showthresholdlimit,
309         a = axis;
310         plot( [a(1) a(2)], [a(1) a(2)].*options.thresholdlimit(1) +
        options.thresholdlimit(2), '-', 'DisplayName', 'Threshold Limit' );
311     end;
312 end;
313 if options.showbandsinlegend || options.showbestfits ||
options.showthresholdlimit || options.showsegmentsbestfit,
314     [legendHandle, lineHandle] = legend( 'show', 'Location', 'best' );
315     if options.showbandsinlegend,
316         set( lineHandle( (1:colourSteps)*2 + numel(lineHandle)/3 ),
        'MarkerSize', 20 );
317     end;
318     if options.scalethreshold~=0 && ~isempty(o),
319         set( lineHandle( colourSteps*2 + 2 + numel(lineHandle)/3 ),
        'MarkerSize', 20 );
320     end;
321 end;
322 hold off;

324 disp( fiterrors );

326 if ~strcmp( options.save, '' ),

```



```

327 %     saveas((gcf, options.save, 'png' );
328 %     print('-dpng', [options.save '.png']);
329 print( fig, '-dpng', [options.save '.png'], '-r150'); %, ['-f' num2str(gcf)]
330 end;

332 function [colours] = getColoursBetween( lowColour, highColour, steps, useHSL
    )
333 %PLOTGRAPHDATA:GETCOLOURSBETWEEN Tweens between two colours in the
334 % given number of steps using either RGB or HSL colour spaces.
335 % If the number of steps is one, the lowColour is returned.
336 if ( steps == 1 )
337     colours = lowColour;
338 else
339     if useHSL,
340         lowColour = rgb2hsl(lowColour);
341         highColour = rgb2hsl(highColour);
342     end;
343     c = (0:1/(steps-1):1)';
344     colours = (1-c) * lowColour + c * highColour;
345     if useHSL,
346         colours = hsl2rgb( colours );
347     end;
348 end;
349 end % END GETCOLOURSBETWEEN

351 function [text] = getLabelText( bf, name, isRobust, showRMSE, showMAE )
352 text = [];
353 if isRobust,
354     text = '\newline(Robust Regression';
355 end;
356 if showRMSE,
357     if isempty( text ),
358         text = '\newline(';
359     else
360         text = [text, ', '];
361     end;
362     if strcmpi( name, 'depths' ) || strcmpi( name, 'depthsum' ) ||
        strcmpi( name, 'segments' ),
363         text = [text 'RMSE = ' num2str( round(bf.rmse) ) ];
364     else
365         text = [text 'RMSE = ' num2str( round(bf.rmse.(name)) ) ];
366     end;
367 end;
368 if showMAE,
369     if isempty( text ),
370         text = '\newline(';
371     else
372         text = [ text, ', ' ];
373     end;
374     if strcmpi( name, 'depths' ) || strcmpi( name, 'depthsum' ) ||
        strcmpi( name, 'segments' ),
375         text = [ text 'MAE = ' ...
376                 num2str( round(bf.mae) ) ];
377     else
378         text = [ text 'MAE = ' ...
379                 num2str( round(bf.mae.(name)) ) ];
380     end;
381 end;
382 if ~isempty( text ),
383     text = [ text ')' ];
384 end;
385 if strcmpi( name, 'depths' ) || strcmpi( name, 'depthsum' ) || strcmpi(
    name, 'segments' ),
386     text = strcat( bf.equation, text );
387 else

```

```
388         text = strcat( bf.equation.(name), text );
389     end;
391 end
392 end %END PLOTGRAPHDATA
```

rgb2hsl.m and hsl2rgb.m are available from:

<http://www.mathworks.com/matlabcentral/fileexchange/20292>

E.4.9 PlotNumberOfElements.m

```

1 function PlotNumberOfElements( data, t, varargin )
2 % PLOTNUMBEROFELEMENTS Plots the number of elements in a data set.

4 options = struct( 'type', 'nogc', ...
5                  'facecolour', [.6 .6 .6], ...
6                  'edgecolour', [0 0 0], ...
7                  'textcolour', [0 0 0], ...
8                  'marker', '+', ...
9                  'markersize', 3, ...
10                 'linestyle', 'none', ...
11                 'newfigure', true, ...
12                 'yaxis', 0, ...
13                 'save', '', ...
14                 'showasbar', true );

16 options = processOptions( options, 'Graphs:PLOTGRAPHDATA', varargin );

18 switch (data.type)
19     case 'vertex', xcol = 1;
20     case 'edge', xcol = 2;
21     case 'root', xcol = 3;
22     otherwise, warning( 'Graphs:PLOTNUMBEROFELEMENTS', ...
23                         'Unknown Data Type: Assuming vertex' );
24     xcol = 1;
25 end;

27 switch options.type,
28     case 'nogc', x = data.data.nogc(:, xcol);
29     case 'gc', x = data.data.gc(:, xcol);
30     case 'error', x = data.data.error(:, xcol);
31     case 'full', x = [data.data.nogc(:, xcol); data.data.gc(:, xcol)];
32     otherwise, warning( 'Graphs:PLOTNUMBEROFELEMENTS', ...
33                         'Unknown Type: Assuming nogc' );
34     x = data.data.nogc(:, xcol);
35 end;

37 if options.newfigure,
38     screenSize = get(0,'ScreenSize');
39     figure( 'Position', [screenSize(3)/6 screenSize(4)/6 ...
40                         2*screenSize(3)/3 2*screenSize(4)/3], ...
41            'PaperSize', [20.98/3 29.68/5], ...
42            'Color', 'white');
43 end;

45 num = grpstats( x, x, 'numel' );

47 hold on;

49 if options.showasbar,
50     un = unique( num );
51     nnum = grpstats( num, num, 'numel' );
52     percent = 100*nnum/sum(nnum);
53     bar( un, percent, 'BarWidth', 0.5, 'FaceColor', options.facecolour, ...
54          'EdgeColor', options.edgecolour );
55     title( t );
56     ylabel( 'Frequency (%)' );
57     xlabel( 'Number of Data Points' );
58     if options.yaxis > 0,
59         axis( [min(un)-0.5 max(un)+0.5 0 options.yaxis] );
60     end;
61     for n=1:numel(un),
62         text( un(n), percent(n) + .05 * max(percent), ...

```

```

63         ['(' num2str( nnum(n) ) ')'], 'Color', options.textcolour, ...
64         'HorizontalAlignment', 'center' );
65     end;
66     set( gca, 'xtick', min(un):max(un) );
67     set( gca, 'box', 'off' );
68 else
69     ux = unique(x);
70     bf = fitdata2( ux, num, { { 'x2', 'x' } } );
71     plot( ux, num, 'Marker', options.marker, 'Color', [0 0 1], ...
72           'LineStyle', options.linestyle, 'MarkerSize', options.markersize, ...
73           'DisplayName', 'Frequency' );
74     plot( ux, bf.data.x2_x, 'r-', ...
75           'DisplayName', [bf.equation.x2_x ...
76           '\newline MAE = ' num2str(bf.mae.x2_x)] );
77     title( t );
78     ylabel( 'Frequency' );
79     switch (xcol)
80     case 1, xlabel( 'Number of Vertices' );
81     case 2, xlabel( 'Number of Edges' );
82     case 2, xlabel( 'Root Vertex ID' );
83     end;
84     legend( 'show', 'Location', 'southoutside' );
85     a = axis;
86     axis( [a(1) a(2) max(a(3), 0) min(a(4), 100)] );
87 end;

89 if ~strcmp( options.save, '' ),
90     saveas((gcf, options.save, 'png' );
91 end;
92 hold off;

94 end

96 function [str] = formatNumber( n )
97     if n < 0,
98         str = num2str(n);
99     else
100         str = ['+' num2str(n)];
101     end;
102 end

```

E.4.10 PlotShowCodeWarmUp.m

```

1 function PlotShowCodeWarmUp(data, t, varargin)
2 %PLOTSHOWCODEWARMUP(data,t, varargin)
3 % data: Array of Data [vertices edges root time gcCount]
4 % t: Title for the graph

5
6 options = struct( 'semilogy',          false, ...
7                  'ylim',              -1, ...
8                  'ypercentile',        -1, ...
9                  'usesubplot',         false, ...
10                 'showbetween',        [data.data.nogc(1,3) ...
11                                         data.data.nogc(1,3)], ...
12                 'showerrors',          false );

14 options = processOptions( options, 'Graphs:PLOTGRAPHDATA', varargin );

16 switch (data.type)
17     case 'vertex', xcol = 1;
18                   column = 'Vertex';
19     case 'edge',   xcol = 2;
20                   column = 'Edge';
21     case 'root',   xcol = 3;
22                   column = 'Root';
23     otherwise,     warning( 'Graphs:PLOTGRAPHDATA', ...
24                             'Unknown Data Type: Assuming vertex' );
25                   xcol = 1;
26                   column = 'Vertex';
27 end;

29 if options.showbetween(1) == options.showbetween(2),
30     t = [t ' (' column ' = ' num2str(options.showbetween(1))];
31     nogc = removeoutliers( data.data.nogc, xcol, 4, ...
32                             'notequaltox', options.showbetween(1) );
33     if options.showerrors,
34         gc = removeoutliers( [data.data.gc; ...
35                               [data.data.error zeros(size(data.data.error, 1),1)] ...
36                               ], xcol, 4, 'notequaltox', options.showbetween(1) );
37     else
38         gc = removeoutliers( data.data.gc, xcol, 4, ...
39                             'notequaltox', options.showbetween(1) );
40     end;
41 else
42     t = [t ' (' column ' = ' num2str(options.showbetween(1)) '-' ...
43         num2str(options.showbetween(2))];
44     nogc = removeoutliers( data.data.nogc, xcol, 4, ...
45                             'notbetweenx', options.showbetween );
46     if options.showerrors,
47         gc = removeoutliers( [data.data.gc; ...
48                               [data.data.error zeros(size(data.data.error, 1),1)] ...
49                               ], xcol, 4, 'notbetweenx', options.showbetween );
50     else
51         gc = removeoutliers( data.data.gc, xcol, 4, ...
52                             'notbetweenx', options.showbetween );
53     end;
54 end;

55 end;
56 if options.showerrors,
57     t = [t ', Errors Shown'];
58 end;
59 screenSize = get(0,'ScreenSize');
60 figure( 'Position', [screenSize(3)/6 screenSize(4)/6 ...
61                     2*screenSize(3)/3 2*screenSize(4)/3], ...
62         'PaperType', 'a4letter', ...

```

```

63         'PaperSize',    [20.98 29.68], ...
64         'Color',        'white');

65
66 if ( options.usesubplot )
67     subplot(2,1,1), semilogy( nogc(:,6), nogc(:,4), 'b+', ...
68                               gc(:,6), gc(:,4), 'r+', 'MarkerSize', 3 );
69     % Create title
70     title( [t ''] );
71     % Create xlabel
72     xlabel( 'Repetition Number' );
73     % Create ylabel
74     ylabel( 'Time (ns)' );
75     % Remove bounding box
76     set( gca, 'box', 'off' );
77 end;

78
79 if options.ypercentile > 0,
80     nogc = removeoutliers( nogc, xcol, 7, ...
81                             'greaterthany', options.ypercentile );
82     gc    = removeoutliers( gc, xcol, 7, ...
83                             'greaterthany', options.ypercentile );
84     t = [t ' ', Time <= ' num2str(options.ypercentile) 'th Percentile'];
85 end;

86 if options.ylimit > 0,
87     nogc = removeoutliers( nogc, xcol, 4, ...
88                             'greaterthany', options.ylimit );
89     gc    = removeoutliers( gc, xcol, 4, ...
90                             'greaterthany', options.ylimit );
91     t = [t ' ', Time <= ' num2str(options.ylimit) 'ns'];
92 end;

93
94 % Plot data
95 if ( options.semilogy )
96     if ( options.usesubplot )
97         subplot(2,1,2), semilogy( nogc(:,6), nogc(:,4), 'b+', gc(:,6), gc(:,4),
98                                   'r+', 'MarkerSize', 3 );
99     else
100         semilogy( nogc(:,6), nogc(:,4), 'b+', gc(:,6), gc(:,4), 'r+',
101                   'MarkerSize', 3 );
102     end;
103 else
104     if ( options.usesubplot )
105         subplot(2,1,2), plot( nogc(:,6), nogc(:,4), 'b+', gc(:,6), gc(:,4),
106                               'r+', 'MarkerSize', 3 );
107     else
108         plot( nogc(:,6), nogc(:,4), 'b+', gc(:,6), gc(:,4), 'r+', 'MarkerSize',
109               3 );
110     end;
111 end;

112
113 % Create title
114 title( [t ''] );
115 % Create xlabel
116 xlabel( 'Repetition Number' );
117 % Create ylabel
118 ylabel( 'Time (ns)' );
119 % Create legend
120 legend( { 'Uninterrupted by GC', 'Interrupted by GC' }, 'Location', 'Best' );
121 % Remove bounding box
122 set( gca, 'box', 'off' );

```

E.4.11 processOptions.m

```

1 function [options] = processOptions( options, fnctName, userArguments )
2
3 for k = 1:2:numel( userArguments ),
4     if ischar( userArguments{k} ) && isfield( options, lower( userArguments{k} )
5     ),
6         opt = options.(lower( userArguments{k} ));
7         if k + 1 <= numel( userArguments ),
8             if strcmp( class( opt ), class( userArguments{k + 1} ) ),
9                 switch class( opt ),
10                     case { 'struct', 'logical', 'char', 'cell' },
11                         options.(lower( userArguments{k} )) = userArguments{k+1};
12                     otherwise,
13                         if size( opt, 1 ) == size( userArguments{k+1}, 1 ) && ...
14                             size( opt, 2 ) == size( userArguments{k+1}, 2 ),
15                             options.(lower( userArguments{k} )) =
16                                 userArguments{k+1};
17                         else
18                             warning( fnctName, ...
19                                     [ 'Invalid option: ' ...
20                                       userArguments{k} ' should be size ' ...
21                                       num2str( size( opt, 1 ) ) 'x' ...
22                                       num2str( size( opt, 2 ) ) ] );
23                         end;
24                     end;
25                 else
26                     switch class( opt ),
27                         case 'struct', warn = 'struct';
28                         case 'logical', warn = 'logical';
29                         otherwise, warn = [ class( opt ) ' [' ...
30                                             num2str( size( opt, 1 ) ) 'x' ...
31                                             num2str( size( opt, 2 ) ) ']' ];
32                     end;
33                     warning( fnctName, ...
34                             [ 'Invalid option: ' userArguments{k} ...
35                               ' should be type ' warn ] );
36                 end;
37             else
38                 warning( fnctName, ...
39                         [ 'Not enough parameters: ' userArguments{k} ] );
40             end;
41         else
42             warning( fnctName, [ 'Invalid Option: ' userArguments{k} ] );
43         end;
44     end;
45 end;

```

E.4.12 removeoutliers.m

```

1 function [output, varargout] = removeoutliers(data, xcol, ycol, type, limit)
2 % Function to remove outliers
3
4 switch lower( type ),
5     case 'lessthanx',          outlying = data(:, xcol) < limit;
6     case 'lessthanx',          outlying = data(:, ycol) < limit;
7     case 'lessthanorequalx',   outlying = data(:, xcol) <= limit;
8     case 'lessthanorequaly',   outlying = data(:, ycol) <= limit;
9     case 'greaterthanx',       outlying = data(:, xcol) > limit;
10    case 'greaterthanx',       outlying = data(:, ycol) > limit;
11    case 'greaterthanorequalx', outlying = data(:, xcol) >= limit;
12    case 'greaterthanorequaly', outlying = data(:, ycol) >= limit;
13    case 'equaltox',            outlying = data(:, xcol) == limit;
14    case 'equaltoy',            outlying = data(:, ycol) == limit;
15    case 'notequaltox',         outlying = data(:, xcol) ~= limit;
16    case 'notequaltoy',         outlying = data(:, ycol) ~= limit;
17    case 'notbetweenx',         outlying = (data(:, xcol) < limit(1) | ...
18                                data(:, xcol) > limit(2));
19    case 'notbetweeny',         outlying = (data(:, ycol) < limit(1) | ...
20                                data(:, ycol) > limit(2));
21    case 'notwithinx',          outlying = (data(:, xcol) <= limit(1) | ...
22                                data(:, xcol) >= limit(2));
23    case 'notwithiny',          outlying = (data(:, ycol) <= limit(1) | ...
24                                data(:, ycol) >= limit(2));
25    case 'morethansdfrommeanx', d = data(:, xcol);
26                                s = abs(d - mean( d )) / std( d );
27                                outlying = s > limit;
28                                clear d s;
29    case 'morethansdfrommeany', d = data(:, ycol);
30                                s = abs(d - mean( d )) / std( d );
31                                outlying = s > limit;
32                                clear d s;
33    case 'greaterthanmeany',    x = data(:, xcol);
34                                y = data(:, ycol);
35                                ux = unique(x);
36                                outlying = zeros(size(x));
37                                filter = grpstats(y,x,'mean') + limit;
38                                for i=1:numel( filter )
39                                    outlying = outlying + (x == ux(i)).*(y >
40                                        filter(i));
41                                end;
42                                clear x y filter ux;
43    case 'greaterthanpercentile',
44                                x = data(:, xcol);
45                                y = data(:, ycol);
46                                ux = unique(x);
47                                outlying = zeros(size(x));
48                                filter = groupedpercentiles([x y], limit);
49                                for i=1:numel( filter )
50                                    outlying = outlying + (x == ux(i)).*(y >
51                                        filter(i));
52                                end;
53                                clear x y filter ux;
54    case 'notismemberx',        outlying = ~ismember( data(:, xcol), limit );
55    case 'notismembery',        outlying = ~ismember( data(:, ycol), limit );
56    case 'ismemberx',           outlying = ismember( data(:, xcol), limit );
57    case 'ismembery',           outlying = ismember( data(:, ycol), limit );
58    case 'greaterthanlinear',    outlying = data(:, ycol) > (data(:, xcol) .*
59                                limit(1) + limit(2));
60    otherwise,                  outlying = [];
61 end;

```



```
60 output = data;  
61 output(any(outlying, 2), :) = [];  
62 if nargout > 1,  
63     inverse = data;  
64     inverse(any(~outlying, 2), :) = [];  
65     varargout{1} = inverse;  
66 end;
```

General Mathematical Notations

Name	Description	Pg.
$\wedge$	BOOLEAN AND: The statement $f \wedge g$ is true if both f and g are true otherwise it is false.	460
$\vee$	BOOLEAN OR: The statement $f \vee g$ is true if either f or g (or both) are true otherwise it is false.	460
$/$	NEGATION: A slash through a symbol denotes the inverse of that symbol's meaning. I.e. $\neq$ denotes not equals; $\notin$ denotes the "is not an element of" relationship; and $\nexists x \in X : f(x)$ denotes that no element x exists in the set X such that $f(x)$ is true.	460
$\lceil x \rceil$	CEILING: Round x up to the nearest integer.	460
$\lfloor x \rfloor$	FLOOR: Round x down to the nearest integer.	460
$\text{round}(x)$	ROUND: Round x to the nearest integer.	460
$ x $	CARDINALITY: The cardinality (size) of set x .	460
$\in$	SET MEMBERSHIP: $x \in X$ denotes that x is a member of the set X .	460
$\subseteq$	SUBSET: $X \subseteq Y$ denotes that every element of X is also an element of Y .	460
$\subset$	PROPER SUBSET: $X \subset Y$ denotes $X \subseteq Y \wedge X \neq Y$.	460
$\cap$	SET INTERSECTION: $A \cap B$ denotes the set of those elements which are contained in both A and B .	460
$\setminus$	SET COMPLEMENT: $A \setminus B$ denotes the set of those elements which are contained in A that are not in B .	460
$\cup$	SET UNION: $A \cup B$ denotes the set of those elements which are contained either in A , or in B , or in both.	460
$\exists$	EXISTS: $\exists x \in X : f(x)$ denotes that at least one element x exists in the set X such that $f(x)$ is true.	460
$\exists!$	UNIQUELY EXISTS: $\exists! x \in X : f(x)$ denotes that exactly one element x exists in the set X such that $f(x)$ is true.	460
$\forall$	FOR ALL: $\forall x \in X : f(x)$ denotes that every element x in the set X is such that $f(x)$ is true.	460
$\equiv$	EQUIVALENCE RELATIONSHIP: $f \equiv g$ denotes that f and g have an equivalence relationship.	460
$\{\dots\}$	SET: An unordered collection of distinct objects. A set can be defined with the notation: $\{\alpha, \beta, \gamma, \dots\}$ denoting an (unordered) collection of the listed elements. $\{x \in X : f(x)\}$ denoting the set containing each element x from the super-set X where $f(x)$ is true. - For example: $\{n \in \mathbb{N}_1 : n \leq 5\} = \{3, 1, 4, 2, 5\}$.	460

Name	Description	Pg.
$\emptyset$	EMPTY SET: A set containing no elements.	460
$\langle \dots \rangle$	SEQUENCE: A set of elements paired with a total order over those elements. A sequence can be defined with the notation: • $\langle \alpha, \beta, \gamma, \dots \rangle$ denoting a sequence containing the elements, and in the order, as listed. • $\langle \mathcal{S}, \mathcal{R} \rangle$ denoting a sequence containing the elements in set $\mathcal{S}$ and the relationship $\mathcal{R}$ which induces a total order over $\mathcal{S}$. • For example: $\langle \{n \in \mathbb{N}_1 : n \leq 5\}, > \rangle = \langle 5, 4, 3, 2, 1 \rangle$.	460

Graph Notations

Name	Description	Pg.
UPPER CASE LETTER	<i>Typically denotes a collection (set, sequence, graph, etc.) of elements.</i>	
$\mathcal{E}$	EDGES, SET OF: The set $\mathcal{E}$ of edges of the graph.	28
$\mathcal{E} \setminus \mathcal{F}$	COTREE EDGES, SET OF: The set $\mathcal{E} \setminus \mathcal{F}$ of edges not contained in the spanning tree of the graph as defined by a Trémaux Tree.	28
$\mathcal{F}$	TREE EDGES, SET OF: The set $\mathcal{F}$ of edges contained in the spanning tree of the graph as defined by a Trémaux Tree.	28
$\mathcal{G}$	GRAPH (SHORT FORM): Shortened form of $\mathcal{G}(\mathcal{V}, \mathcal{E})$.	28
$\mathcal{G}(\mathcal{V}, \mathcal{E})$	GRAPH: A (connected) graph with set $\mathcal{V}$ of vertices and the set $\mathcal{E}$ of edges.	28
$\mathcal{T}$	TRÉMAUX TREE (SHORT FORM): Shortened form of $\mathcal{T}(\mathcal{G}, \mathcal{F}, r)$.	28
$\mathcal{T}(\mathcal{G}, \mathcal{F}, r)$	TRÉMAUX TREE: A Trémaux Tree $\mathcal{T}$ explicitly defined, for graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, as partitioning the edges into the set $\mathcal{F}$ of tree edges (and the set $\mathcal{E} \setminus \mathcal{F}$ of cotree edges) and where vertex r is the root of the tree.	28
$\mathcal{V}$	VERTICES, SET OF: The set $\mathcal{V}$ of vertices for the graph.	28
LOWER CASE LETTER	<i>Typically denotes a single element/object (number, vertex, edge, etc.).</i>	
r	ROOT VERTEX: The root vertex of a Trémaux Tree.	28
EDGES		
$\{u, v\}$	EDGE: An undirected edge connecting vertices u and v .	28
$(u \rightarrow v)$	DIRECTED EDGE: An directed edge outbound from vertex u and inbound to vertex v .	28
$(u \xrightarrow{\mathcal{T}} v)$	TREE EDGE: A directed edge that is also a tree edge.	28
$(u \xrightarrow{\mathcal{C}} v)$	COTREE EDGE: A directed edge that is also a cotree edge.	28
INTERVALS		
$[u; v]$	CHAIN: An ordered closed interval of graph elements from minimum u to maximum v , inclusive, such that all elements are related, and ordered, via $\prec$ and that the chain is maximal with regards to set inclusion. $[u; v] = \langle \{x \in \mathcal{V} \cup \mathcal{E} : u \preceq x \preceq v\}, \prec \rangle$	31
$]u; v]$	CHAIN EXCLUDING MINIMUM: $]u; v] = \langle \{x \in \mathcal{V} \cup \mathcal{E} : u \prec x \preceq v\}, \prec \rangle$	31
$[u; v[$	CHAIN EXCLUDING MAXIMUM: $[u; v[= \langle \{x \in \mathcal{V} \cup \mathcal{E} : u \preceq x \prec v\}, \prec \rangle$	32
$]u; v[$	CHAIN EXCLUDING MINIMUM AND MAXIMUM: $]u; v[= \langle \{x \in \mathcal{V} \cup \mathcal{E} : u \prec x \prec v\}, \prec \rangle$	32
RELATIONS		
$\preceq$	PRECEDES OR EQUALS: $u \preceq v$ if, and only if, u is contained in the path from r to v within the Trémaux Tree.	28

Name	Description	Pg.
$\prec$	PRECEDES: $u \prec v$ if, and only if, $u \preceq v \wedge u \neq v$.	28
$\prec_{\text{IM}}$	IMMEDIATELY PRECEDES: $u \prec_{\text{IM}} v$ if, and only if, $u \prec v$ and no element x exists such that $u \prec x \prec v$.	28
$\perp$	COMPARABLE PRECEDENCE: $u \perp v \Leftrightarrow u \preceq v \vee v \preceq u$.	28
EDGE RELATIONS		
$\prec^*$	TT PRECEDENCE (PARTIAL ORDER): $(u \rightarrow v) \prec^* (u \rightarrow w) \Leftrightarrow (\text{lowPoint1}((u \rightarrow v)) \prec \text{lowPoint1}((u \rightarrow w))) \vee$ $(\text{lowPoint1}(e_1) = \text{lowPoint1}(e_2) \wedge \text{lowPoint2}(e_1) = v \wedge \text{lowPoint2}(e_2) \prec$	30
$\parallel^*$	INCOMPARABLE TT-PRECEDENCE: $(u \rightarrow v) \parallel^* (u \rightarrow w) \Leftrightarrow (\text{lowPoint1}((u \rightarrow v)) = \text{lowPoint1}((u \rightarrow w))) \wedge$ $((u = \text{lowPoint2}((u \rightarrow v)) = \text{lowPoint2}((u \rightarrow w))) \vee$ $(\text{lowPoint2}((u \rightarrow v)) \prec u \wedge \text{lowPoint2}((u \rightarrow w)) \prec u))$	30
$<^{**}$	TT PRECEDENCE (TOTAL ORDER): $(u \rightarrow v) <^{**} (u \rightarrow w) \Leftrightarrow (((u \rightarrow v) \prec^* (u \rightarrow w)) \wedge ((u \rightarrow v) \parallel^* (u \rightarrow w))$ and $(u \rightarrow v)$ is arbitrarily ordered before $(u \rightarrow w))$	30
SEGMENTS		
$<^s$	SEGMENT PRECEDENCE: A total order over the set of segments.	38
$\text{head}(s)$	HEAD: The head (minimal $\prec$ precedence) element of segment s .	32
$\text{headEdge}(s)$	HEAD EDGE: The edge with minimal $\prec$ precedence within segment s .	32
$\text{headVertex}(s)$	HEAD VERTEX: The vertex immediately preceding the head edge for segment s .	32
$\text{tail}(s)$	TAIL: The tail (maximal $\prec$ precedence) element of segment s .	32
$\text{tailEdge}(s)$	TAIL EDGE: The edge with minimal $\prec$ precedence within segment s .	32
$\text{tailVertex}(s)$	TAIL VERTEX: The vertex the tail edge for segment s is inbound to.	32
e_h^s	HEAD EDGE (COMPACT NOTATION): The edge with minimal $\prec$ precedence within segment s .	41
v_h^s	HEAD VERTEX (COMPACT NOTATION): The vertex immediately preceding the head edge for segment s .	41
e_t^s	TAIL EDGE (COMPACT NOTATION): The edge with minimal $\prec$ precedence within segment s .	41
v_t^s	TAIL VERTEX (COMPACT NOTATION): The vertex immediately preceding the tail edge for segment s .	41
$v_{l_1}^s$	FIRST LOW POINT VERTEX (COMPACT NOTATION): The minimal precedence vertex reachable from segment s ($v_{l_1}^s = \text{lowPoint1}(\text{head}(s))$).	41
$v_{l_2}^s$	FIRST LOW POINT VERTEX (COMPACT NOTATION): The second lowest precedence vertex reachable from segment s ($v_{l_2}^s = \text{lowPoint2}(\text{head}(s))$).	41
EMBEDDINGS		
Γ	EMBEDDING: An embedding; assumed to be a planar embedding unless otherwise stated.	40
$\mathcal{M}$	SURFACE: A surface (orientable, genus 0, two-dimensional topological manifold) onto which an embedding is performed.	40
$\mathcal{F}$	FACES: The set $\mathcal{F}$ of faces defined by an embedding Γ within a surface $\mathcal{M}$ is such that the union of all faces is the complement of the embedding $(\mathcal{M} \setminus \Gamma)$ and that each face is a maximal set of connected points within that complement.	40

Glossary

Name	Description	Symbol	Pg.
adjacent	In the abstract, two distinct vertices are adjacent if they are connected by an edge. Specifically relating to embeddings – an element (vertex or edge) is adjacent to a face if the embedding of that element is on the boundary of that face. Similarly, a segment is adjacent to a face (or region) if any element contained in that segment is adjacent to that face (or region).		40
ancestor	A given segment's ancestors are those segments containing an element which precedes an element of that given segment (or, simply, segment s_α is an ancestor of segment s_β if $\text{head}(s_\alpha) \prec \text{head}(s_\beta)$). The (root) segment containing the root vertex of the Trémaux Tree has no ancestors but is an ancestor of every other segment.		33
articulation vertex	An articulation vertex is a boundary vertex in a 1-separation (such that deletion of that articulation vertex separates the connected component containing that vertex forming two disconnected connected components).		
biconnected component	A biconnected component is a maximal 2-separable (2-connected) sub-graph.		36
biconnected graph	A biconnected graph is a graph such that if any element (vertex or edge) of that graph is removed then all other elements of that graph remain connected. A biconnected graph is k -separable such that $k \geq 2$.		465
block	A block of segments is a group of segments, related via conflicts, which all have embeddings within either the inside or outside region adjacent to a common ancestor where the embedding of any one segment from that block within one of those regions defines the embedding of each other segment of that block to be in either the same or opposite region.		50

Name	Description	Symbol	Pg.
branch	A branch, of the Trémaux Tree of graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, rooted at an element (vertex or edge) x is the sub-tree $\mathcal{G}'(\mathcal{V}', \mathcal{E}')$ containing only those elements preceded by, or equalling, x ; i.e. such that $(\forall v \in \mathcal{V}' : v \in \mathcal{V} \wedge x \preceq v)$ and $(\forall e \in \mathcal{E}' : e \in \mathcal{E} \wedge x \preceq e)$. The branch rooted at the root of a Trémaux Tree is the entire Trémaux Tree.		28
bridge edge	A bridge edge of a graph is an edge that is not contained in any biconnected component of that graph. Deletion of a bridge edge separates the component containing that edge into two disconnected components. All bridge edges are tree edges (see corollary 4.1.21) and their adjacent vertices are articulation vertices.		36
chain	A chain is an ordered sequence of graph elements (a path) such that: within the Trémaux Tree, all elements of that chain are related, and ordered, via $\prec$; and that the chain is maximal with regards to set inclusion.		31
child	A given segments children are the set of segments whose parent is that given segment.	$[\alpha; \beta]$	33
component	A k-separable component is a maximal k-separable sub-graph. • A component (without qualifier), or connected component, indicates a 1-separable component. • A biconnected component indicates a 2-separable component (and is a sub-graph of a 1-separable component). • A triconnected component indicates a 3-separable component (and is a sub-graph of a biconnected component).		
conflict	A conflict (or a planarity conflict) occurs when two segments do not allow a planar embedding within the same region (either inside or outside) relative to a common ancestor.		50
connected graph	A graph such that a path exists connecting every pair of elements within that graph. A connected graph is k -connected such that $k \geq 1$. See also: biconnected graph.		467
cotree edge	A cotree edge $(u \xrightarrow{\mathcal{C}} v)$ of $\mathcal{G}$ is an edge that is not contained in the spanning tree defined by the Trémaux Tree of $\mathcal{G}$.	$(u \xrightarrow{\mathcal{C}} v) \in \mathcal{E} \setminus \mathcal{T}$	28
curve	Topologically, a curve γ is a continuous mapping $\gamma : \mathcal{I} \rightarrow \mathcal{P}$ from one-dimensional (an interval $\mathcal{I}$ of real numbers) to n -dimensional space (a continuous path $\mathcal{P}$).	γ	466

Name	Description	Symbol	Pg.
curve, closed	A closed curve γ is where the mapping $\gamma : \mathcal{J} \rightarrow \mathcal{P}$ is such that letting $\mathcal{J} = [a; b]$ then $\gamma(a) = \gamma(b)$; i.e. the curve has no end-points and completely encloses an area.		40, 466
curve, open	An open curve γ is where the mapping $\gamma : \mathcal{J} \rightarrow \mathcal{P}$ is such that letting $\mathcal{J} = [a; b]$ then $\gamma(a) \neq \gamma(b)$; i.e. the curve has two distinct end-points.		40, 466
curve, simple	A simple open curve γ is such that the mapping $\gamma : \mathcal{J} \rightarrow \mathcal{P}$ is injective – i.e. the curve does not intersect itself. A simple closed curve $\gamma : [a; b] \rightarrow \mathcal{P}$ is such that $[a; b[\rightarrow \mathcal{P}$ is injective and that $\gamma(a) = \gamma(b)$ – i.e. the curve forms a loop which does not intersect itself.		
cycle	A cycle is a closed path (where the first and last element of the path are the same and that each vertex of the cycle is incident to an even number of edges also contained in that cycle).		466
cycle, simple	A simple cycle is a cycle where each vertex of the cycle is incident to exactly two edges of the cycle; i.e. the cycle forms a single loop that does not intersect itself. Each simple cycle's planar embedding forms a Jordan curve.		
degree	The degree of a vertex is the number of edges incident to that vertex.		
descendant	A given segment's descendants are those segments containing an element which succeeds an element of that given segment (or, simply, segment s_α is a descendant of segment s_β if $\text{head}(s_\beta) \prec \text{head}(s_\alpha)$). Every non-root segment is a descendant of the root segment.		33
edge	A connection between a distinct pair of vertices – if looping edges are allowed then edges can connect a non-distinct pair of vertices.	$\{u, v\} \in \mathcal{E}$	467
element	Each element of a graph is a fundamental unit (vertex or edge) of that graph.		29
embedding	An embedding of a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ on a surface $\mathcal{M}$ is a mapping from each vertex of $\mathcal{G}$ to a point on $\mathcal{M}$ and from each edge to a open curve (or, for a looping edge, a closed curve) such that the end-points of the curve are at the points corresponding to the incident vertices.		40

Name	Description	Symbol	Pg.
embedding, planar	A planar embedding Γ of a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ is an embedding on a surface $\mathcal{M}$ (assumed to be orientable and genus 0) such that each vertex maps to a <i>distinct</i> point on $\mathcal{M}$ and each edge corresponds to a Jordan arc (or Jordan curve for looping edges) such that no arc (curve) intersects with itself (by definition of it being a Jordan arc/curve), another arc nor a point corresponding to a vertex, except to terminate at the points corresponding to its incident vertices.	Γ	40
face	A face f is a maximal set of connected points within the complement $(\mathcal{M} \setminus \Gamma)$ of an embedding Γ on a surface $\mathcal{M}$.	$f \in \mathcal{F}$	40
graph	A graph $\mathcal{G}$ is an ordered pair $(\mathcal{V}, \mathcal{E})$ consisting of a set $\mathcal{V}$ of vertices and a set $\mathcal{E}$, disjoint from $\mathcal{V}$, of edges, together with an incidence function $\psi_{\mathcal{G}}$ that associates with each edge of $\mathcal{G}$ an unordered pair of (not necessarily distinct) vertices of $\mathcal{G}$. If e is an edge and u and v are vertices such that $\psi_{\mathcal{G}}(e) = \{u, v\}$ then e is said to <i>connect</i> u and v , and the vertices u and v are said to be <i>incident</i> to e and to <i>terminate</i> e . The numbers of vertices and edges in $\mathcal{G}$ can be denoted by $ \mathcal{V} $ and $ \mathcal{E} $, respectively. All graphs are assumed to be connected graphs unless otherwise stated.	$\mathcal{G}$	28
incident	A vertex v and an edge $\{a, b\}$ are incident if, and only if, $v = a \vee v = b$. A segment s is incident to a vertex v if v is not contained in s but is incident to an edge (either the head or tail edge) contained in s .		42
Jordan arc	A Jordan arc is a simple open curve.		40
Jordan curve	A Jordan curve is a simple closed curve.		40
k -separable	A graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, is k -separable (or k -connected) if to separate $\mathcal{G}(\mathcal{V}, \mathcal{E})$ into two disconnected components requires a minimum of k distinct vertices to be removed from that graph (i.e. there exists a k -separation and there does not exist a $k - 1$ -separation).		467
k -separation	For a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, a k -separation is a partition on the edge-set $\mathcal{E}$ of $\mathcal{G}$ into the sub-sets $\mathcal{E}'$ and $\mathcal{E} \setminus \mathcal{E}'$ such that the sub-graphs induced by both edge-sets are non-empty and connected and that there are exactly k boundary vertices common to both incident sub-graphs. An articulation vertex is the boundary vertex for a 1-separation. See also: k -separable.		12

Name	Description	Symbol	Pg.
leaf	A leaf of a Trémaux Tree is an element where no other element succeeds it (i.e. a vertex with no outbound edges or a cotree edge). Each leaf terminates a branch of the Trémaux Tree and its containing segment.		34
low point vertex, first	A given element's first low point vertex is the minimum precedence vertex of that given element's low point vertices.		29
low point vertex, second	A given element's second low point vertex is the second minimum precedence vertex of that given element's low point vertices. If the element only has a single low point vertex then that low point vertex is both the first and second low point vertex.		29
low point vertices	A given element's low point vertices are those vertices which precede or equal that given element (within the Trémaux Tree) and are incident to edges contained in the sub-Trémaux Tree rooted at that given element.		29
microsecond	One millionth (10^{-6}) of a second.	μs	104
millisecond	One thousandth (10^{-3}) of a second.	ms	104
nanosecond	One thousand millionth (10^{-9}) of a second.	ns	104
parent	A given segment's parent is its ancestor which contains the highest precedence element preceding that given segment (or, simply, segment s_α is the parent of the distinct segment s_β if $\text{headVertex}(s_\beta) \in s_\alpha$). The root segment has no parent; every other segment has a parent.		33
path	A path is a sequence $\langle e_1, e_2, \dots, e_n \rangle$ of distinct edges, corresponding to a sequence $\langle v_0, v_1, v_2, \dots, v_n \rangle$ of vertices incident to those edges, such that $e_i = \{v_{i-1}, v_i\}$; i.e. each given edge of the path has one incident vertex which is also incident to the preceding edge of the sequence (assuming it is not the first edge of the sequence) and its other incident vertex is also incident to the succeeding edge of the sequence (assuming it is not the last edge of the sequence).		
path, closed	A closed path is a path where the elements terminating the path are identical; i.e. the path forms a cycle.		466
path, open	An open path is a path where the elements terminating the path are distinct; i.e. the path does not form a cycle.		
path, simple	A simple path is a path where no strict sub-path of that path is a cycle; i.e. no more than two edges on a simple path are incident to any one vertex also on that path.		

Name	Description	Symbol	Pg.
reachable	A given element's reachable elements are those elements contained in the sub-tree of the Trémaux Tree rooted at that given element and any vertices incident to edges contained in that sub-tree (which may be contained in that sub-tree or precede it).		29
region (relative to a segment)	A region relative to a segment is defined by a fixed set of points corresponding to a face adjacent to that segment when that segment is embedded. Embeddings of subsequent segments may bisect that region, forming smaller faces contained within that region (which, themselves, are regions relative to those subsequent segments), however that region is constant. • Each Class 1 segment has exactly one adjacent region. • Each Class 2, 3 or 4 segment has exactly two adjacent regions – designated as inside and outside.		40
root	The vertex designated to be the root of the Trémaux Tree and, hence, having minimal $\prec$ precedence.	r	28
root (of a branch of a Trémaux Tree)	The element (vertex or edge) having minimal $\prec$ precedence within the branch of the Trémaux Tree.		
rooted spanning tree	A spanning tree where one vertex is designated as the root vertex. A rooted spanning tree can be given direction such that all edges are directed relative to the root vertex (typically all edges are directed away from the root vertex).		28
segment	A path of $\prec$ related vertices and/or edges with specific properties (see page 32). Each connected graph can be partitioned into a set of disjoint segments.		32
segment, root	The root segment is the segment containing the root vertex of the Trémaux Tree; it has no ancestor (hence, no parent or siblings) and each other segment is one of its descendants.		33
sibling	Two segments are siblings if they share the same parent.		33
spanning tree	A sub-graph of a given connected graph which is a tree that covers all vertices of that given graph.		469
sub-graph	A sub-graph of a given graph is a graph composed of a (strict or non-strict) sub-set of the elements from that given graph.		

Name	Description	Symbol	Pg.
surface	A two-dimensional topological manifold. In the context of this work, all surfaces are assumed to be orientable and have genus 0; i.e. the $\mathbb{R}^2$ plane or the surface of a sphere.		40
tree	A connected graph containing no looping edges or cycles.		469
tree edge	A tree edge $(u \xrightarrow{T} v)$ of $\mathcal{G}$ is an edge contained in the spanning tree defined by the Trémaux Tree of $\mathcal{G}$.	$(u \xrightarrow{T} v) \in \mathcal{F}$	28
Trémaux Tree	A Trémaux Tree $(\mathcal{T}(\mathcal{G}, \mathcal{F}, r))$ of graph $(\mathcal{G}(\mathcal{V}, \mathcal{E}))$ defines a root vertex r and a partitioning of the set $\mathcal{E}$ of edges of $\mathcal{G}$ into two disjoint sets: the set $\mathcal{F}$ of tree edges which compose a rooted spanning tree of $\mathcal{G}$ (rooted at r); and the set $\mathcal{E} \setminus \mathcal{F}$ of cotree edges. All edges (tree and cotree) of the Trémaux Tree connect a pair of vertices which are comparable via the $\preceq$ relationship.	TT	28
vertex	A fundamental unit of a graph.	$v \in \mathcal{V}$	467

Graphical Notations

Name	Description	Pg.
Vertex		471
Vertex (and optional descendants)	 A vertex and (optional) chains of descendants forming a (partial) sub-tree; this is represented graphically by an oversized vertex entity. <i>Note: A tree edge represented as being inbound to this graphical entity is inbound to the minimum precedence vertex rooting the sub-tree; any other (outbound or inbound cotree) edge incident to this graphical entity can be incident to any vertex contained within the represented sub-tree – where two such incident edges can be within different branches of the sub-tree.</i>	471
Root Vertex		471
Undirected Edge		471
Tree Edge		471
Cotree Edge		471
Chain (non-empty)	 A chain containing one or more edges.	471
Chain (possibly empty)	 A chain containing zero or more edges.	471

Bibliography

- [1] T. W. Anderson. “On the Distribution of the Two-Sample Cramer-von-Mises Criterion”. In: *Annals of Mathematical Statistics* 33.3 (1962), pp. 1148–1159.
- [2] L. Auslander and S. V. Parter. “On Imbedding Graphs In The Plane”. In: *Journal of Mathematics and Mechanics* 10.3 (1961). (<http://www.iuj.indiana.edu/IUMJ/ISSUE/1961/10/10032>, Last Accessed: 18/04/2011), pp. 517–523.
- [3] Norman L. Biggs, E. Keith Lloyd, and Robin J. Wilson. *Graph Theory 1736-1936*. Oxford: Oxford University Press, 2006. ISBN: 978-0-19-853916-2.
- [4] J.A. Bondy and U.S.R. Murty. *Graph Theory*. 3rd Corrected Printing. Vol. 244. Graduate Texts in Mathematics. Springer, 2008, p. 654. ISBN: 978-1-84628-969-9.
- [5] K. S. Booth and G. S. Lueker. “Testing the Consecutive Ones Property, Interval Graphs, and Graph Planarity using PQ-Tree Algorithms”. In: *Journal of Computing System Science* 13 (1976), pp. 335–379.
- [6] Brent Boyer. *Robust Java Benchmarking - Understand the Pitfalls of Benchmarking Java Code*. <http://download.boulder.ibm.com/ibmdl/pub/software/dw/java/j-benchmark1-pdf.pdf> and <http://download.boulder.ibm.com/ibmdl/pub/software/dw/java/j-benchmark2-pdf.pdf>. (Last Accessed 06/02/2010). 2008.
- [7] John M. Boyer. “Additional PC-Tree Planarity Conditions”. In: *Graph Drawing*. Vol. 3383. Lecture Notes in Computer Science. 2005, pp. 82–88.
- [8] John M. Boyer and Wendy Myrvold. “Stop Minding Your P’s and Q’s: a Simplified O(N) Planar Embedding Algorithm”. In: *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*. 1999, pp. 140–146.
- [9] John M. Boyer and Wendy Myrvold. *Simplified O(N) Planarity Algorithms*. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.8.6613>. 2001.
- [10] John M. Boyer and Wendy Myrvold. “Stop Minding Your P’s and Q’s: Simplified Planarity by Edge Addition”. In: *Journal of Graph Algorithms and Applications* (2003), p. 25.
- [11] John M. Boyer et al. *Correcting and Implementing the PC-tree Planarity Algorithm*. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.9.243>. 2003.
- [12] John M. Boyer et al. “Lempel, Even, and Cederbaum Planarity Method”. In: *Proceedings of the 3rd Workshop on Experimental Algorithms*. Vol. 3059. Lecture Notes in Computer Science. 2004, pp. 129–144.
- [13] John M. Boyer et al. “Stop Minding Your P’s and Q’s: Implementing a Fast and Simple DFS-Based Planarity Testing and Embedding Algorithm.” In: *Proceedings of the 11th International Conference on Graph Drawing 2003*. Lecture Notes in Computer Science 2912 (2004). Ed. by G. Liotta, pp. 25–36.
- [14] N. Chiba et al. “A Linear Algorithm for Embedding Planar Graphs using PQ-Trees.” In: *Journal of Computer and Systems Sciences* 30 (1985), pp. 54–76.

- [15] Hubert de Fraysseix. “Trémaux Trees and Planarity”. In: *Electronic Notes in Discrete Mathematics* 31 (2008), pp. 169–180.
- [16] Hubert de Fraysseix and Patrice Ossona de Mendez. *Public Implementation of a Graph Algorithm Library and Editor* (PIGALE). <http://pigale.sourceforge.net/>. 2008.
- [17] Hubert de Fraysseix, Patrice Ossona de Mendez, and Pierre Rosenstiehl. “Depth-First Search and Planarity”. In: *International Journal of Foundations of Computer Science* 17.5 (2006), pp. 1017–1030.
- [18] Hubert de Fraysseix and P. Rosenstiehl. “A Depth-First Search Characterization of Planarity”. In: *Annals of Discrete Mathematics* 13 (1982), pp. 75–80.
- [19] Hubert de Fraysseix and P. Rosenstiehl. “A Characterization of Planar Graphs by Trémaux Orders”. In: *Combinatorica* 5.2 (1985), pp. 127–135.
- [20] M. Demoucron, Y. Malgrange, and R. Petruiset. “Graphes planaires: Reconnaissance et construction de représentations planaires topologiques”. In: *Rev. Française Recherche Opérationnelle* 8 (1964), pp. 33–47.
- [21] N Deo. “Note on Hopcroft and Tarjan planarity algorithm.” In: *Journal of the Association of Computing Machinery* 23.1 (Jan. 1976). (<http://dx.doi.org/10.1145/321921.321929>, Last Accessed: 18/04/2011), pp. 74–75.
- [22] Leonhard Euler. “Solutio problematis ad geometriam situs pertinentis”. In: *Commentarii Academiae Scientiarum Imperialis Petropolitanae* 8 (1736), pp. 128–140.
- [23] Shimon Even and Robert Endre Tarjan. “Computing an *st*-numbering”. In: *Theoretical Computer Science* 2.3 (1976), pp. 339–344.
- [24] Eclipse Foundation. *Eclipse.org*. <http://www.eclipse.org/>. 2010.
- [25] J. A. Gallian. “Dynamic Survey DS6: Graph Labeling.” In: *Electronic Journal of Combinatorics* DS6 (2007). (<http://www.combinatorics.org/Surveys/ds6.pdf>, Last Accessed: 18/04/2011), pp. 1–58.
- [26] A. J. Goldstein. “An Efficient and Constructive Algorithm for testing whether a Graph can be embedded in a Plane”. In: *Graph and Combinatorics Conference*. Office of Naval Research Logistics Project, Department of Mathematics, Princeton University, 1963.
- [27] Branko Grünbaum and G. C. Shephard. “Convex Polytopes”. In: *Bulletin of the London Mathematical Society* 1.3 (1969), pp. 257–300.
- [28] H. Guggenheimer. “The Jordan Curve Theorem and an Unpublished Manuscript by Max Dehn”. In: *Archive for History of Exact Science* 17.2 (1977), pp. 193–200.
- [29] Bernhard Haeupler and Robert E. Tarjan. “Planarity Algorithms via PQ-Trees (Extended Abstract)”. In: *TGGT 2008 international conference: Topological and Geometric Graph Theory*. (Conference presentation: <http://www.canalc2.tv/video.asp?idVideo=7543&voir=oui>). 2008.
- [30] Thomas C. Hales. “Jordan’s Proof of the Jordan Curve Theorem”. In: *Studies In Logic, Grammar and Rhetoric* 10.23 (2007), pp. 45–60.
- [31] J. E. Hopcroft and R. E. Tarjan. “Planarity Testing in $V \log V$ steps: Extended Abstract”. In: *Proceedings of the IFIP Cong. (1971): Foundations of Information Processing* (1971), pp. 18–22.
- [32] John E. Hopcroft and Robert Endre Tarjan. “Efficient Planarity Testing”. In: *Journal of Associated Computing Machinery* 21.4 (1974), pp. 549–568.
- [33] Wen-Lian Hsu. “PC-Trees vs. PQ-Trees”. In: *Lecture Notes in Computer Science* 2108 (2001).
http://www.iis.sinica.edu.tw/IASL/hsu/webpdf/paper-2001-PC-Trees_vs._PQ-Trees.pdf, pp. 207–217.

- [34] Wen-Lian Hsu. *An Efficient Implementation of the PC-Tree Algorithm of Shih & Hsu's Planarity Test*. Tech. rep.
http://www.iis.sinica.edu.tw/IASL/hsu/webpdf/paper-2003-PLANAR_implementation.pdf.
Institute of Information Science, Academia Sinica, 2003.
- [35] Wen-Lian Hsu and R. M. McConnell. "PC Trees and Circular-Ones Arrangements". In: *Theoretical Computer Science* 296.1 (2003), pp. 99–116.
- [36] Wen-Lian Hsu and R. M. McConnell. "Handbook of Data Structures and Applications". In: ed. by Dinesh P. Mehta and Sartaj Sahni.
http://www.iis.sinica.edu.tw/IASL/hsu/webpdf/paper-2003-PQ_Trees_PC_Trees_&_Planar_Graphs.pdf.
Chapman and Hall, 2004. Chap. PQ Trees, PC Trees and Planar Graphs. ISBN: 1584884355.
- [37] Frederick James. *Statistical Methods In Experimental Physics*. 2nd. Singapore: World Scientific Publishing Co. Pte. Ltd., 1988. ISBN: 981-256-795-X.
- [38] Selmer M. Johnson. "Generation of Permutations by Adjacent Transpositions." In: *Math. Comput.* 17 (1963), pp. 282–285.
- [39] Camille Jordan. *Cours d'Analyse de l'École Polytechnique*. 2nd (1893). 1887.
- [40] David Joyner. *Adventures in group theory: Rubik's cube, Merlin's machine, and other mathematical toys*. 2nd. Baltimore: John Hopkins University Press, 2008. ISBN: 978-0-8018-9013-0.
- [41] William Kocay. *The Hopcroft-Tarjan Planarity Algorithm*.
<http://web.archive.org/web/20030504193618/http://bkocay.cs.umanitoba.ca/G&G/articles/Planarity.pdf>.
(Last Accessed: 20/01/2011). 1993.
- [42] Eriola Kruja et al. "A Short Note on the History of Graph Drawing". In: *Proceedings of Graph Drawing 2001*. Vol. 2265. Springer LNCS, 2001, pp. 272–286.
- [43] Eriola Kruja et al. "A Short Note on the History of Graph Drawing". In: *Graph Drawing*. Vol. 2265. Springer Berlin / Heidelberg, 2002, pp. 602–606.
- [44] C. Kuratowski. "Sur le problème des courbes gauches en topologie". In: *Fundamental Mathematics* 5 (1930), pp. 217–283.
- [45] Lucien Le Cam and Grace Lo Yang. *Asymptotics in Statistics: Some Basic Concepts*. 2nd. Springer Series in Statistics. Springer, Aug. 2000. ISBN: 978-0387950365.
- [46] W. G. Leibniz. "Letter to Christiaan Huygens, September 8, 1679". In: *Vorstudien zur Topologie*. Ed. by I. Gerhardt and J. B. Listing. Vol. 1. Gottinger Studien, 1847, pp. 811–875.
- [47] A. Lempel, S. Even, and I. Cederbaum. "An Algorithm for Planarity Testing of Graphs". In: *Theory of Graphs: International Symposium, Rome 1966*. Ed. by P. Rosenstiehl. Gordon and Breach, New York, 1967, pp. 215–232.
- [48] Henry H. Liu. *Software Performance and Scalability: A Quantitative Approach*. Ed. by Larry Bernstein. (Wiley Series on Quantitative Software Engineering). Wiley-Blackwell, June 2009. ISBN: 978-0-470-46253-9.
- [49] Yanpei Liu. "Modulo 2 Programming and Planar Embeddings". In: *Acta Math. Appl. Sinica* 1 (1978), pp. 321–329.
- [50] Yanpei Liu. "On Boolean Characterizations of Planarity and Planar Embeddings of Graphs". In: *Annals of Operations Research* 24 (1990), pp. 165–174.
- [51] Édouard Lucas. *Récréations Mathématiques*. 2nd (1891). Vol. 1. Paris: Gauthier-Villars Et Fils, 1882.
- [52] K. Mehlhorn and P. Mutzel. "On the Embedding Phase of the Hopcroft and Tarjan Planarity Testing Algorithm". In: *Algorithmica* 16 (1996), pp. 233–242.
- [53] K. Mehlhorn, P. Mutzel, and St. Näher. *An Implementation of the Hopcroft and Tarjan Planarity Test and Embedding Algorithm*. Tech. rep. MPI I 93 151. Max Planck Institut für Informatik, 1993.

- [54] K. Mehlhorn and St. Näher. “LEDA: A library of efficient data types and algorithms”. In: *CACM* 38.1 (1995), pp. 96–102.
- [55] Lee F. Mondshien. *Combinatorial Ordering and the Geometric Embedding of Graphs*. Tech. rep. DTIC Accession No.: AD0732882. Lincoln Laboratory, M.I.T., Aug. 1971.
- [56] Harold James Ruthren Murray. *A History of Board-Games other than Chess*. Oxford University Press, Dec. 1952. ISBN: 978-0198274018.
- [57] P. Mutzel. *A Fast Linear Time Embedding Algorithm Based on the Hopcroft-Tarjan Planarity Test*. Tech. rep. Institut für Informatik, Universität zu Köln, 1992.
- [58] W. F. Osgood. *Lehrbuch der Funktionentheorie*. Teubner, 1912.
- [59] H. Poincaré. “Sur la généralisation d’un théorème d’Euler relatif aux polyèdres.” In: *Comptes rendus hebdomadaires des séances de l’Académie des Sciences* 117 (1893), pp. 144–145.
- [60] William H. Press et al. *Numerical Recipes: The Art of Scientific Computing*. 3rd. Cambridge University Press, Sept. 2007. ISBN: 978-0521880688.
- [61] Helen Purchase. “Which aesthetic has the greatest effect on human understanding?” In: *Proceedings of Graph Drawing Symposium*. Vol. 1353. Lecture Notes in Computer Science. Springer-Verlag, 1997.
- [62] Helen C. Purchase. “Metrics for Graph Drawing Aesthetics”. In: *Journal of Visual Languages and Computing* 13 (2002), pp. 501–516.
- [63] Helen C. Purchase, Jo-Anne Alder, and David Carrington. “Graph Layout Aesthetics in UML Diagrams: User Preferences”. In: *Journal of Graph Algorithms and Applications* 6.3 (2002), pp. 255–279.
- [64] Edward M. Reingold, Jurg Nievergelt, and Narsingh Deo. *Combinatorial Algorithms: Theory and Practice*. Prentice Hall College Div, 1977. ISBN: 978-0131524477.
- [65] Neil Robertson and P. D. Seymour. “Graph Minors. XVI. Excluding a non-planar graph”. In: *Journal of Combinatorial Theory, Series B* 89.1 (2001), pp. 43–76.
- [66] Neil Robertson and P. D. Seymour. “Graph minors. XX. Wagner’s conjecture”. In: *Journal of Combinatorial Theory, Series B* 92.2 (2004), pp. 325–357.
- [67] P. Rosenstiehl. “Preuve Algebrique Du Critere De Planarite De Wu-Liu”. In: *Annals of Discrete Mathematics* 9 (1980), pp. 67–78.
- [68] Wei-Kuan Shih and Wen-Lian Hsu. “A Simple Test For Planar Graphs”. In: *Proceedings of the International Workshop on Discrete Mathematics and Algorithms*. http://www.iis.sinica.edu.tw/IASL/hsu/webpdf/paper-1993-A_simple_test_for_planar_graphs.pdf. 1993, pp. 110–122.
- [69] Wei-Kuan Shih and Wen-Lian Hsu. “A New Planarity Test”. In: *Theoretical Computer Science* 223 (1999). http://www.iis.sinica.edu.tw/IASL/hsu/webpdf/paper-1999-A_New_Planarity_test.pdf, pp. 179–191.
- [70] M. A. Stephens. “Use of the Kolmogorov-Smirnov, Cramer-Von Mises and Related Statistics Without Extensive Tables”. In: *Journal of the Royal Statistical Society. Series B (Methodological)* 32.1 (1970), pp. 115–122.
- [71] Sun Microsystems. *Java API Documentation - java.lang.System*. Feb. 2010.
- [72] R. Tarjan. “An Efficient Planarity Algorithm”. STAN-CS-244-71. PhD thesis. Computer Science Dept., Stanford University, Nov. 1971.
- [73] R. Tarjan. “Depth-First Search and Linear Graph Algorithms”. In: *SIAM J. Comput.* 1.2 (June 1972), pp. 146–160.
- [74] Gaston Tarry. “Le Problème Des Labyrinthes”. In: *Nouvelles Annales de Mathématiques* 3.14 (1895), pp. 187–190.

- [75] Robin Thomas. *Planarity in Linear Time*. <http://www.math.gatech.edu/~thomas/planarity.ps>. (Last Accessed: 20/01/2011). Dec. 1995, Revised 1997.
- [76] H. F. Trotter. “Perm (Algorithm 115)”. In: *Communications of the ACM* 5 (1962), pp. 434–435.
- [77] Oswald Veblen. “Theory on plane curves in non-metrical analysis situs”. In: *Transactions of the American Mathematical Society* 6 (1905), pp. 83–98.
- [78] Klaus Wagner. “Über eine Eigenschaft der ebenen Komplexe”. In: *Mathematische Annalen* 114.1 (1937), p. 570.
- [79] Klaus Wagner. *Graphentheorie*. Bibliographisches Inst., 1970, p. 220. ISBN: 3411002484.
- [80] Colin Ware et al. “Cognitive Measurements of Graph Aesthetics”. In: *Information Visualization* 1.2 (2002), pp. 103–110.
- [81] Hassler Whitney. “Non-Separable and Planar Graphs”. In: *Transactions of the American Mathematical Society* 34.2 (1932), pp. 339–362.
- [82] Hassler Whitney. “2-Isomorphic Graphs”. In: *American Journal of Mathematics* 55.1 (1933), pp. 245–254.
- [83] S. G. Williamson. “Embedding Graphs in the Plane – Algorithmic Aspects”. In: *Annals of Discrete Mathematics* 6 (1980), pp. 349–384.
- [84] S. G. Williamson. “Depth-First Search and Kuratowski Subgraphs”. In: *Journal of the Association of Computing Machinery* 31.4 (1984), pp. 681–693.
- [85] S. Gill Williamson. *Combinatorics for Computer Science*. Rockville, Maryland: Computer Science Press, 1985. ISBN: 0-88175-020-4.
- [86] Steve Wilson and Jeff Kesselman. *Java™ Platform Performance: Strategies and Tactics*. http://java.sun.com/docs/books/performance/1st_edition/html/JPTitle.fm.html (Accessed: 20/01/2011). Prentice Hall, 2000, p. 256. ISBN: 978-0201709698.
- [87] W. Wu. “A Theory of Imbedding, Immersion and Isotopy of Polytopes in Euclidean Space”. In: *Science Press* (1965).